Software Reuse in Defense Electronics: A Study of Organization and Architecture Approaches in a Challenging Business and Technical Environment

by

Jeffrey Davis

M.S., Computer Science
Worcester Polytechnic Institute, 2001

B.S., Mechanical Engineering, 1996
Cornell University

Submitted to the System Design and Management Program
in Partial Fulfillment of the Requirements for the Degree of

Master of Science in Engineering and Management

at the
Massachusetts Institute of Technology

May 2010
[June 2010]

Signature of Author _____ Jeffrey Davis
System Design and Management Program
May 2010

Certified by _____ Alan MacCormack
Thesis Supervisor
Visiting Associate Professor, Sloan School of Management

Accepted by _____ Patrick Hale
Director
System Design & Management Program

Software Reuse in Defense Electronics: A Study of Organization and Architecture Approaches in a Challenging Business and Technical Environment

by

Jeffrey Davis

Submitted to the System Design and Management Program on 19 May, 2010 in Partial Fulfillment of the Requirements for the Degree of Master of Science in Engineering and Management

Abstract

Although large scale software reuse has been studied and practiced in industry for more than 20 years, there are some practice areas where it has presented both technical and business challenges. A sector notable for exhibiting these challenges is defense electronics, the home business arena of the client organization. We have gathered information from academic and broad industry work to compare with the sponsor's experience over the past 15 years. Their organization has built a software reuse program over this time, and benefits from significant exploration of component, module, and software product line[1] reuse models. In this context, we try to make sense of and understand patterns of the sponsor's cases, also concentrating on the business and technological environment and the resultant constraints that bound software projects. Our general hypothesis is that success of a reuse program is affected by: project organization type, the team's support and performance, and the design-for-reuse quality of the product. The business success that flows from the reuse program is dependent upon the strategic decisions made with reuse in mind as well as the suitability of the program's structure to the overall business model.

In the sponsor's case, this suitability was less than perfect due to the nuances of the defense industry. We draw valuable insights from these cases and present them in a manner useful by similar projects. Further, experience with the product platform technique presents cases that may reveal how it requires the rigor of strict product focus to best serve the business. The main output of this work is to offer conclusions that can be used to shape business area strategy and reuse techniques based on specific conditions of the potential projects or product families.

Thesis Supervisor: Alan MacCormack,
Visiting Associate Professor, Sloan School of Management

[1] http://www.sei.cmu.edu/productlines/index.cfm

Acknowledgements

I would like to gratefully acknowledge the many people that have helped me along the way through this thesis and my time at MIT. First and foremost, I offer a sincere thanks to my company sponsor, Phil Richard. Without his belief in my endeavor and the SDM program, and the support of the Engineering organization and his successors, this educational and enriching experience would not have been possible.

I also offer thanks to the many individuals from the company that have helped with this research, especially: Peter Simonson, Bob Boland, Dave Wardwell, Tom Vaccaro, Ty Robinson, John MacCabe, Jim Wankel, Karen Anderson, Gerard Quintanar, Mike Feeney, Rob Brundage, and Joe Ader. Thanks also to Dan Pikora, John Kelly, and Doug Chabinsky for their patience while I completed this work.

As an academic experience, the program has been enlightening and a great fit for me and my career. In particular, I would like to thank my advisor, Alan MacCormack, for his experience in this field and patience with my work and work style. In general, I acknowledge the positive impact of the MIT experience, enabled by all the students, faculty, staff, and friends that I've interacted with over the past year and a half. For their teaching impact, I would like to thank Michael Davies, Brad Morrison, Olivier de Weck, Bill Aulet, Howard Anderson, Ed Crawley, and Shalom Saar.

Finally, my deepest gratitude goes to my wife, Melinda. Without her support, I wouldn't have made it through this, and she's kept the family running all the while. Time with her and our daughters, Hailey and Caitlyn, makes it all worth it.

Table Of Contents

Index of Figures

Index of Tables

1 Introduction

1.1 Motivation

The development of complex systems in the defense industry is challenging due to the extremely difficult problems the systems must solve simultaneously maintaining viability as a business. Electronics systems exhibit an increasing reliance on software processing to perform the desired capabilities and flexibility. Engineering teams have looked to improve their effectiveness at software development by seeking to reuse earlier bodies of code that were useful. The business of software reuse requires that the development group maintain commonality and product integrity in the customer funded development environment. The sponsoring company would especially like to understand how best to organize a software reuse and product platforming structure in the general business model of their industry.

After spending 15 years working for the sponsor of this research, the author has experienced a full lifecycle of product development, the progression from hardware to software focused implementations, the beginnings of software reuse, and eventually a software product family and a full generation of a product platform approach. The client has seen moderate success starting with early reuse projects that came in over budget and with reuse numbers in the disappointing 50% range. The next wave of projects could not benefit from the lessons of the first group (as they had not completed yet) and were happy to be underbidding competition by 57%. These same efforts would see execution issues after contract award as the reuse library was immature. Eventually, projects would learn to estimate, plan, and execute reuse projects, some seeing the team beat their plan's cost projection by 20% (a cost performance index of 1.2) and reuse of 96%.

Below the highly successful surface of less challenging projects, many serious issues can be found that threaten the viability of the program's future. First, projects with new and challenging technical problems are seeing lower than expected reuse amounts – the library isn't as suitable as it is for some 'more typical' projects. Second, the maintenance

and sustainment funding stream has run out in the defense market's downturn. This challenges the business model of software reuse in this sector. We are motivated to survey the academic and industrial environment for new and relevant techniques, and also take a close look at our experience cases to find unique insights.

1.2 The research question

In general, this work will try to find what framework for software reuse can be prescribed to the client organization. At the highest level of observation over many software intensive projects, it seems that projects can be modeled as having inputs, managerial decisions, execution factors, and resultant outcomes. We are familiar with the array of inputs and constraints that influence the project. The more specific questions below center on guidance that can be found for managerial decisions pertaining to project execution, aiming to yield the most successful outcome possible.

1. What insight does the client's experience with software reuse projects reveal about the impact of input conditions and management decisions on project success?

2. What is a prescriptive framework that will help steer future projects?

1.3 Method

This work begins with a study of prior literature on software reuse for methods and lessons learned. The broader software development industry has applicable lessons to from which we can learn. Of particular value, though, are those insights gained in more closely relevant sectors. To that end, the consulted literature will at least include the following terms: Defense, high performance systems, project organization, software reuse architecture, component development, and development team productivity.

As background, the client's experience will be discussed to illustrate the bigger picture and be at least aware of the system boundaries within which this topic resides. Attempts will be made to extend our study of the problem into as broad a context as possible. Case studies will be presented that highlight the key projects conducted in the client's larger software reuse program.

Data from these projects was then analyzed to understand the project outcomes that may be optimized by strategies and decisions made by management. We will see in section 4.2 which issues tend to have influence on execution and outcome of a reuse effort.

From this research, case study, and analysis, in section 5, we present a prescriptive framework for guidance of future software projects. A graphical view of the problem space is shown here in Figure 1.

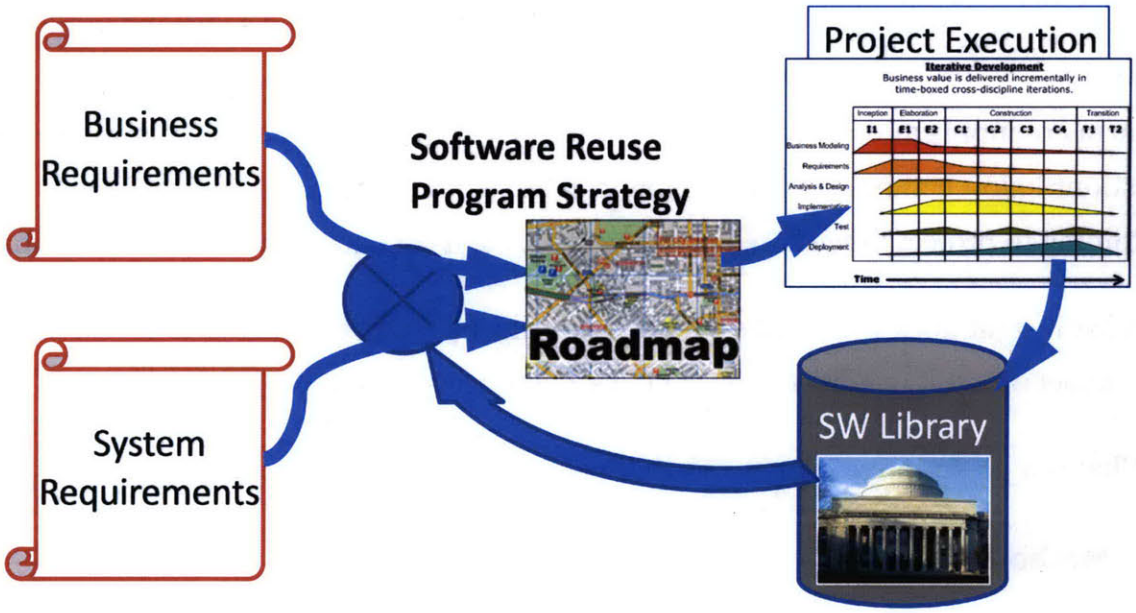


Figure 1: Inputs and Constraints Drive Software Reuse Strategy and Tactics to Enable Successful Execution

The real value of this work is to enable business and project leadership to develop the strategy and roadmap in a more informed manner. The benefit of experience in the software reuse organization is valuable, but it can get scattered and incomplete. This work is an attempt to centralize the experience and insights in a useful and practical way.

2 Literature Review

To begin this study of software reuse, we surveyed the existing literature for everything from broad reuse theory to specific lessons learned from various industries. There were a few works that stand out as solid surveys of software reuse as a general strategy for good engineering practice. What follows is a synopsis of the relevant portions to this research. First, Kreuger's work for Carnegie Mellon University (published in the ACM, 1992) simply titled *Software Reuse* provides us with a good outline for modern techniques of software reuse.

Software reuse has been around since the earliest days of software development. The first developer probably reused his main program constructs for the *second* program he wrote, and the *second* developer probably reused some code from that first developer. Kreuger reminds us that the idea of software engineering was first presented in 1968 as a proposed solution to the *Software Crisis*. McIlroy's thesis at the time was "that the software industry is weakly founded, and that one aspect of this weakness is the absence of a software components sub-industry."[2] The impetus for further work is that despite a long history, the practice of reuse is still fraught with complications that inhibit sustainable, effective reuse efforts. Kreuger presents four techniques for reuse: abstraction, selection, specialization, and integration; and eight approaches, or strategies. The closest match to the sponsor's experience is reuse by integration of source code components[3], where a set of modules communicate through an interconnection framework. In this scenario, the components import and export data and interface in a standard way. The intention is for the components to be the 'unit of reuse'. There are many nuances of this strategy when it is applied to object oriented (OO) programming.

It is worth noting that the sponsor chose an object oriented approach for its software reuse program. There were many reasons for this, not the least of which was that the OO method of software development was at a high point in popularity and buzz around

---

[2] McIlroy, M. D. (1968). Mass Produced Software Components, *Software Engineering Conference for the NATO Science Committee*, 1968.

[3] Kreuger, C. W. (1992). Software Reuse, *ACM Computing Surveys, Vol 24, No 2.* – pg 133

the time of the launch of the reuse program. Many of the principals in the program's design and initial implementation firmly believed in the benefits brought by inheritance, classes, and subclasses. Early versions of the reuse library demonstrated sound execution of these techniques, and boded well for efficient adoption and success of the reuse program. Issues arose as the software library grew with the number of developers. "Overall, the OO benefits were not realized.[4]" Newer developers were not quite as effective at implementing sound OO practice for reuse in the library. The current state of the library exhibits challenges as it conflicts with two of Kreuger's truisms (see inset). The more advanced use of object orientation prevented newer developers from being able to easily 'know what it does' and 'find it faster'.

The following truisms were included earlier in the survey. They are simple statements on reuse that have been difficult to satisfy in practice.

For a software reuse technique to be effective, it must reduce the cognitive distance between the initial concept of a system and its final executable implementation.

For a software reuse technique to be effective, it must be easier to reuse the artifacts than it is to develop the software from scratch.

To select an artifact for reuse, you must know what it does.

To reuse a software artifact effectively, you must be able to "find it" faster than you could "build it."

What are the open areas of research in software reuse? There is clearly much work that remains to be done. In general, the search for high-level abstractions for software artifacts is probably the most crucial, but this type of research is not new or unique to software reuse.

Figure 2: Kreuger's Truisms about Software Reuse

Kreuger also observes that for software reuse in large, complex systems to be effective, abstraction plays a vital role. "Developers must either be familiar with the abstractions a priori or must take time to study and understand the abstractions"[5]. As the client's software reuse library became large and its use spread more widely throughout the organization, more and more developers were employed to execute projects. The core team of developers that created the library was unable to train all the new developers in their standard ways of object oriented programming and the use of the library.

If Kreuger bestows upon us a framework for considering reuse as an objective, later work by Frakes and Terry [6] presents a consolidated summary of models of reuse (ACM,

---

[4] Interview Quote, subject #3
[5] Kreuger, '92, pg 178
[6] Frakes, W. & Terry, C. (1996). Software Reuse: Metrics and Models, *ACM computing surveys, vol 28, no 2*

metrics and models '96) that can be used to determine if the cost of a reuse program is justified by the benefits. This is the next step in the process of determining a software enterprise's reuse strategy. At a summary level, Frakes admits that "none of these models are derived from data, nor have they been validated," though they provide a metric for comparison of potential options. Once a model has been created for multiple options and scenarios, one can use it to explore the limits of a software reuse program's viability.

There are two cost benefit models presented by Gaffney and Durek (1989) that are discussed by Frakes and Terry. They both aim to quantify "the cost of software development for a given product relative to all new code". This means that the result is a factor, "C", representing how much less developing a product with reused code will cost in the long term. A manager can run a few calculations and compare alternatives in the relative cost space. The fact that the models are meant for comparison purposes only, since they are not validated or empirical, limits their value if the user has only a small number of alternatives to explore and little experience with the model. While both models are insightful, the second model incorporates an amortization for the cost of reusability over a number of follow-on projects. This brings in the 'benefit' side of the equation and gives more weight to a project that will be reused more times.

Frakes and Terry's work is a helpful compilation of metrics and models that will enable a team to put most of the relevant issues on the program design table. These issues can be discussed in a objective, metrics based format, which will help identify the best strategy for the reuse program. An example of one model calculation is shown in section 4.4.

Jacobson et al. provide a more broad discussion of the overall reuse concept and how it enables a viable business in "Making the Reuse Business Work"[7]. The most relevant points are:

- A strong architecture is required for a successful reuse program. The structure and interfaces of the software project must be designed so that they will support

[7] Jacobson, I., Griss, M., & Jonsson, P. (1997) Making the Reuse Business Work. *Software Reuse: Architecture, Process and Organization for Business Success*, ACM press, chapter 15 .

reuse while effectively meeting the needs of the system. This means that the software must be extensible enough for long term reuse and modular enough for use by many developers for a portfolio of applications. Jacobson reminds us that an architecture and design that are understandable by the team is important as they go forth and execute projects.

- Instituting a reuse program takes change, which is painful for some individuals. Success of the reuse program requires a strong vision, to make the change work. This is enabled by (and actually requires) an executive commitment. Investment must be made, and the payback period may be lengthy, so high level support is a must.
- The business aspects of a software reuse program are vital. An organization should expect that it will take some time to recover the investment in the reuse program. It is also important that the business model accept the reuse process. "50-70% of business engineering attempts fail, largely due to insufficient attention to the 'soft' factors ...to address these risks with a systematic transition process that emphasizes vision, organization building, focused on competence units, and *suitable models of financing*."[8]

A fundamental work on reuse programs was done by Poulin and Hancock at IBM in the early 1990's. Their paper is a comprehensive look at software reuse it's early days of popularity. It is recommended reading for a larger enterprise seeking to start a reuse program. The lessons learned at IBM coupled with the researcher's combination of technical and business insights lend value to the would-be reuse effort.

Of the main concepts presented as fundamental to reuse, an interesting one is that "software development with reuse" is an entity unto itself. Two approaches are proposed here, composition and generative. In the composition approach, the one employed by the sponsor organization, applications are built out of reusable building blocks. "The building block approach requires components with encapsulated function, well-defined and specified interfaces, and known quality."[9] These building blocks of

---

[8] Jacobson et al (1997)

[9] Poulin, J. S., Caruso, J. M., & Hancock, D. R. (1993). The business case for software reuse, *IBM Systems Journal, 32, 4, pg. 567*

actual code enable a developer to reuse components in a very modular way, and add to them by following the standard interfaces.

Another business unit within the sponsor company uses a generative approach, in which engineers model the software and use "automated tools or generator programs" to produce compile-able code from a model and existing designs. This method is better suited for development teams that are fully trained to model their software, and wish to control the model as well as the code. Both approaches have utility and merit, it just depends on where a team wants to focus its effort – on designs and models, or on code modules. Another relevant point that Poulin makes is in identifying the difference between information (and software) producers and consumers. We will see this is a relevant issue later in our discussion of software reuse by cloning, and then the difficulties of maintaining a single software baseline in a product platform.

Poulin and Hancock break a potential reuse effort down to its cost benefit tradeoffs. They outline some of the models previously discussed (Gaffney and Durek, '89) with an expanded set metrics used by IBM to reflect the effort saved in the reuse program. These metrics are similar to those used by the sponsor's organization to measure their effectiveness as a reuse team. In section 4, we will compile some of these same metrics for the sponsor organization to use in our analysis.

Finally, Poulin and Hancock perform a Net Present Value (NPV) analysis for the Return on Investment (ROI) of a reuse program. In this analysis, they look at Reuse Cost Avoidance (RCA) and Additional Direct Costs (ADC) for the program, both important factors to consider. Figure 5, on pg 591 provides an excellent example of the ROI analysis at the corporate level for a reuse program. The results of this NPV analysis are plotted in Figure 3, below.

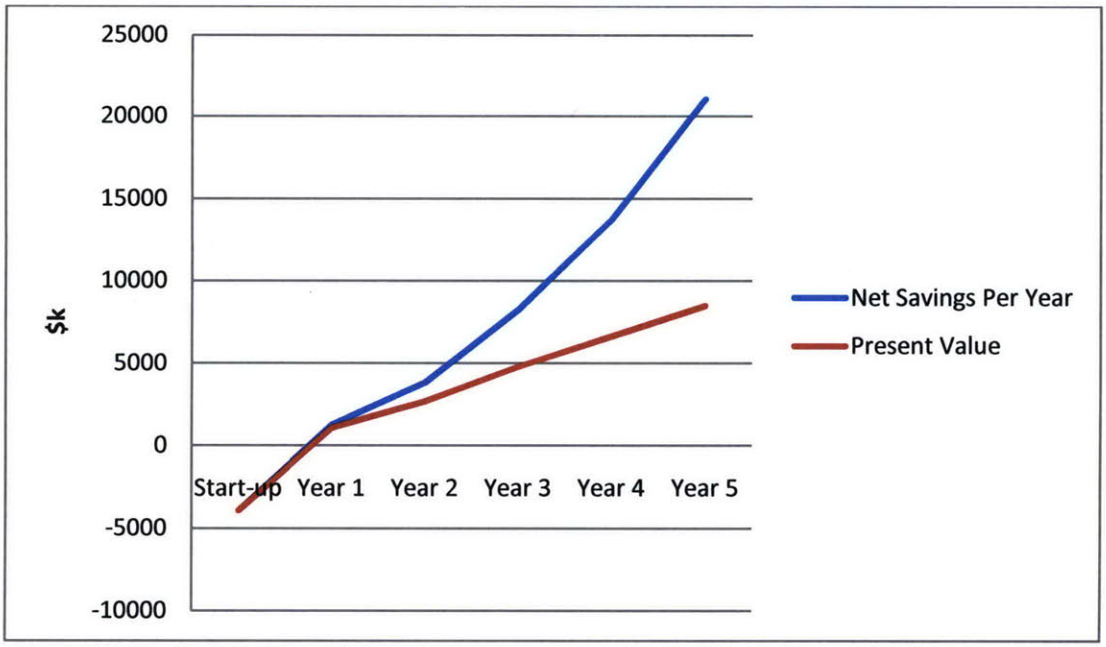


Figure 3: Cost Recovery for a Reuse Case at IBM

The example ROI presented is 104% over the 5 year period. This is "extraordinarily high", but the authors admit that it does not include the inherent risks in many of their underlying assumptions. This is a very insightful point to remember when considering a reuse program. We can make many assumptions on the business case, costs, adoption, reuse percentages, and the like; but they are all predictions and assumptions, each with a high risk of being completely separate from reality. As managers, we must explore the variance of these assumptions and try the cost-benefit analysis out with combinations of less positive situations to see if we are still comfortable with the endeavor.

Fichman and Kemerer 's work from the Journal of Systems and Software, 2001, acknowledges the glut of reuse techniques, literature, and frameworks. Their main insight is that "in spite of this, systematic reuse has been difficult to achieve in practice."[10] This is because programs fail to explore one of the top inhibitors in the success of software reuse programs, the lack of alignment in the motivation of project management with that of the reuse program. The project's cost and schedule constraints do not allow the reuse team to expend the effort to make the software as reusable as they could or should. This is called a lack of "incentive compatibility" –

[10] Fichman, R. G., Kemerer, C. F. (2001). Incentive compatibility and systematic software reuse. *The Journal of Systems and Software, 57, 45-60.*

trying to deliver a product at a competitive cost and schedule is in conflict with the reuse program's objectives of implementing the ideal reuse software components.

Ted Davis, of the Software Productivity Consortium, made an early attempt to create a framework to help evolve a reuse program in 1993. His Reuse Capability Model helps asses an organization's reuse capability and looks at goals and critical success factors for a potential endeavor.

As we will see in the next section, the software reuse program of interest evolved from a component based software reuse effort into a *product platform* approach. This literature review would be incomplete without acknowledgement of the original product platform work set forth by Meyer and Lehnerd in the late '90s. Their definition: "A product platform is a set of subsystems and interfaces that form a common structure from which a stream of derivative products can be efficiently developed and produced."[11] This definition and philosophy guided the development of the sponsor's product platform. There is a significant body of research in the product platform area, from the original book, "The Power of Product Platforms", to any number of interpretive and practical guides. Meyer and Seliger's article applying the topic to software development is a helpful and illustrative guide. Their hypotheses are often upheld, though they advertise success through the product platform technique prior to later longitudinal studies had been conducted that illustrated some sharp potential causes of failure.

Morisio et al. (2002) performed one such study that investigated patterns of success and failure in 25 software reuse projects. Their findings were insightful – highlighting the necessity of executive commitment. Other necessary factors were: introduction of reuse processes, modification of non-reuse processes for reuse, and consideration of human factors[12]. It also cites a misconception that object oriented development and the existence of a reuse repository alone will enable a successful reuse program.

---

[11] Meyer, M. H., Lehnerd, A.P. (1997). The Power of Product Platforms. as quoted in: Meyer and Seliger, R. (1998). *Sloan Management Review*, Fall '98, 61-74.

[12] Morisio, M., Ezran, M., Tully, C. (2002). Success and Failure Factors in Software Reuse, *IEEE transactions on software engineering vol 28, no 4.*

We conclude our literature review with a state of the industry's research by Frakes and Kang. A handful of recent "product line engineering" projects are summarized, with comparative comments for each. The most relevant portion to this study is the identification of sustainability as a challenge. "A current problem is to find the means of sustaining reuse programs on a long term basis"[13]. They highlight the important choice of which parts of a system are reusable and how to do technology transfer. We will see these issues play into the sponsor's software reuse efforts as well.

Much has been studied and written about software reuse that is relevant to our organization's situation and research thereof. The available information on strategies, vital elements of a reuse program, cost-benefit models, and post-mortem studies provides us with enough guidance to create or run a reuse program with some effectiveness. The application of that guidance to our chosen reuse organization is the subject of this research. It is notable that most of the referenced work has been done for the commercial industry, which begs the question, "how is it different in the defense industry?" We will show that the difference in business models creates a gap in the ability to simply transfer all of the available guidance. Our challenge is to narrow that gap and critique the reuse program of interest and provide guidance going forward.

[13] Frakes, W., Kang, K. (2005). Software Reuse Research, Status and Future. *IEEE transactions on Software engineering vol 31, no 7, 529-536.*

3 Sponsor experience with software reuse

3.1 Definitions and Background

The previous sections of this work dealt with motivation for software reuse (in a challenging business and technical environment) and past literature that was relevant to the same topic. Throughout those sections, we have used the terms project, program, and product heavily. Before we cover details of these with respect to the client's business, let us provide definitions for these entities.

Project: An individual software development effort. These efforts have their own requirements, budget, and schedule constraints, but can be part of a family of projects all for a similar customer.

Reuse Program: A concerted effort to reuse software throughout an organization on multiple projects. Good (but not all) reuse programs are choreographed from the individual developers through management to executive sponsorship. They also include the development and business processes of the organization. They aim to be systematic, effective, and cost saving.

Product: A system that the organization develops and/or sells. For this study, the system has significant software content.

Product Line: A lineage of related products. Newer ones are based on their predecessors, and they all have some common functionality. The Software Engineering Institute at Carnegie Mellon defines: *"A software product line (SPL) is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.*[14]"

We will discuss software reuse as it applies to the defense electronics sector. This discussion includes the defense acquisition process, of which there are many phases. The official depiction of this process is shown below in Figure 4.

[14] http://www.sei.cmu.edu/productlines/index.cfm

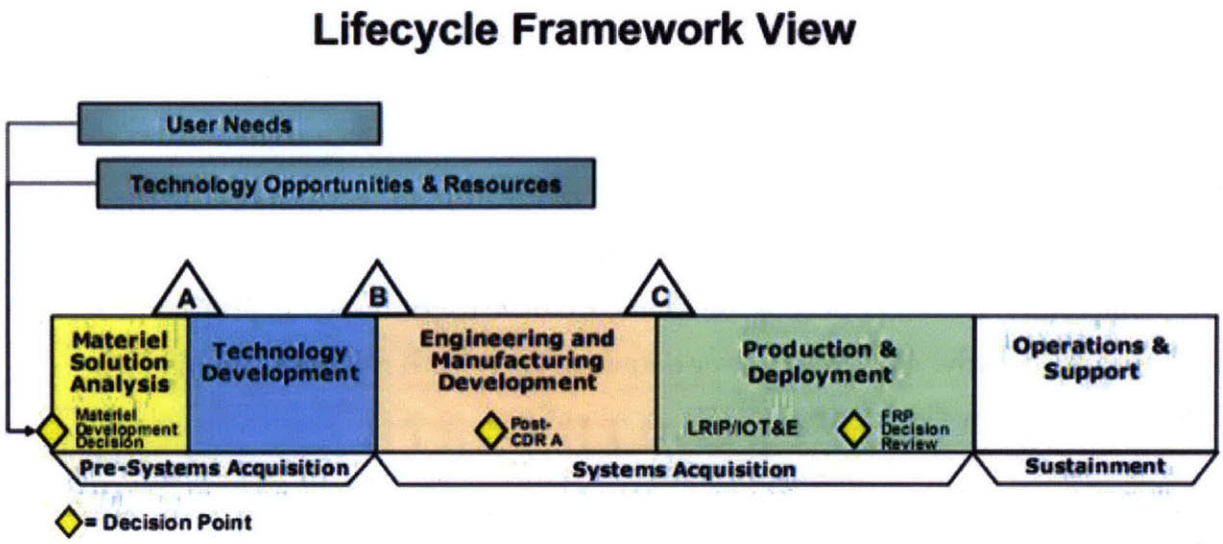


Figure 4: The DoD acquisition process and lifecycle[15]

From the contractor's perspective, there is also a parallel process that overlaps with "User Needs" and "Pre-Systems Acquisition" where new business ideas are cultivated and a candidate solution is developed. Some definitions of these steps are below:

New Business Capture: A broad term referring to the process of developing new business, such as competing in proposals, up to and including the signing of a contract.

Technical Baseline: The technical "approach" taken (or planned) by the development team to create a system. This includes the system hardware, and software designs, however rough they may be early on in a project.

System Architecture: Though there are many definitions, one is: *"The embodiment of concept, and the allocation of physical/informational function to elements of form, and definition of interfaces among the elements and with the surrounding context."* [16] In a product line or product platform, many systems will use a 'common' architecture.

Implementation (Code, Unit Test): The phase of development where real code is written. It begins after the architecture and design are set, and ends with a unit test of the software.

---

[15] https://acc.dau.mil/CommunityBrowser.aspx?id=332375&lang=en-US
[16] Prof. E. Crawley, MIT ESD.34 Lecture Notes, Lecture 1, Fall 2009.

Integration and Test: The phase of development where individual functional pieces (subsystem 1, 2, hardware, software) are brought together and made to work as a system. It also includes verifying that the integrated system meets its requirements.

In Figure 4, we don't see the detail on most of these phases other than new business capture. The others occur in varying rigor and duration during each of the phases after milestone A.

3.2 A brief history of the subject Software product family

Systems of electronics have been procured and used by the US military since WWI. The cold war saw a major uptick in development of electronics systems as techniques used by our adversaries became too complex to defeat 'by hand'. These systems were primarily hardware based until general computing and software development became a mainstream approach in the 1980's. Technology then evolved to host software on military grade hardware and/or ruggedized commercial computing hardware for military use. The reconfigurability and lower size, weight, and power of software systems made them a more attractive basis around which to form a solution, and they gained in prominence. By the mid 1990s, one could expect to visit certain portions of Navy Ships and Air Force platforms and see racks and racks of electronics all running software to enable the warfighter to do his/her job.

This study's client participated in that rise of electronics and software systems. The mid 1990s saw major projects executed through development to production and deployment of systems into the user's hands. At the same time, the evolution to object oriented programming and the rise in popularity of reconfigurable software architectures brought many competitors to offer a flexible, expandable (or so they advertized) software solution to new materiel procurement requirements. The industry-wide realization was that "now (circa '99) we can [build a system] in software where we couldn't before."[17] The client researched these ideas, but took no substantive action. The combination of these factors caused more than one large competitive acquisition to be awarded to another contractor over the client, despite the client's incumbent status in

[17] Interview quote, subject #1

that space. This caused a major wake-up call and a serious re-grouping throughout the client's leadership and technical ranks. A reconfigurable, reusable software product of their own was adopted as the solution, and so began the long process of implementing this solution.

"Large Scale software reuse was the whole idea," commented the software family's technical director, who is commonly referred to as the 'father' of the program. In this case, the technical director was in the perfect position to lay out the main idea; his responsibilities were for both technology strategy and direction, while driven primarily by the new business development effort. What the leadership was hearing from their customers at the time was that they "needed a modular architecture in software (similar to modularity in hardware) to enable hardware independence and scalability. The 3-4 tiered model that the architecture team developed would require a fast communications fabric, but also one that was very robust. The team would spend days and weeks trying to "stump" themselves as to how a type of system could not be built from their architecture.

Once the architecture was set, a crucial step according to Jacobson ('97), a team of the organization's finest software developers and algorithmists embarked on building the core reuse modules. They would become a software product family, that we will call Odin. It required a "really strong vision to ride herd" on the creative development team, who were just as likely to stray from the vision as any team when faced with a challenging task for an extended period of time. At the outset, the main group of developers "didn't know well how to design for reuse", but the program's leadership was simultaneously out reading everything they could find on the subject and bringing their learnings back to the team. These sources included the relatively new Rational Unified Process (RUP) and its associated reuse efforts[18]. RUP would serve as the model most closely followed by Odin's implementation team. One might observe that a reuse library creation team that included algorithmists might not be ideal – the core software should be developed with only reuse in mind. The rationale for including them on the team was that in this competitive environment, if the initial development effort did not emerge

[18] Jacobson et al. (1997).

with a demonstrable system that performed core domain functions, the investment would be a failure. The leadership team felt they had to parlay the resulting prototypes into a contract effort as soon as possible, and customer relevance would be therefore required at the outset.

Throughout this process, the motivation was clear:

- The business needed a solution to rival competitor's software solutions that were being chosen by the finite customer set.
- Everyone believed in one of the tenants of software reuse, "why should doing *essentially the same thing*" ever need to be *re*done, especially if it had been done well once already?
- If a solid reuse library existed, time would be saved by not re-coding existing software and funding would be available to work on the technology improvements and system quality that the customers and taxpayers paid for and expected out of the client's teams.
- As software became the backbone of all the electronics systems developed, the size of the software effort became the metric upon which all other functions based their scope of work. If the software effort could be reduced, it had a multiplicative effect on the effort and costs saved throughout the project.

By 2000, the software product family was ready for its first demonstration projects. The model for reuse was examined and the organization chose a version of the component approach (according to Poulin et al pg 569), that of course fit their component-framework architecture. In the first iterations, the reuse library required executive support to convince a new start project to adopt it as their baseline. The project groups (usually related by customer) were tightly knit and found it difficult to accept that their new project would be created by a nucleus of software developed by 'some other group' on internal research funding. In order to let these projects maintain their identity and cohesiveness, and for the reuse program team to continue development of the library, the organization created 'client projects'. These projects would be located in their existing physical space and staffed mostly by engineers already on related or legacy projects. They would take a release from the reuse program and use the base

components and communications framework to create their system. The reuse program staff would be 'on call' to support issues as they arose. This became known as the 'client project' model, and was not without its share of issues as development got underway. We will discuss these issues in the case subsection, to follow.

As the reuse program continued, with the mixed success of the first client projects and the resultant growth of the "client support" role for the reuse team, the enterprise realized that if that support role was to continually increase, the apparent next step was a full-fledged support organization that was all but required to enable success by the client projects. The difficulties the projects had with immaturity of the library and applying it to their system demonstrated that the software wasn't "really all that portable."[19] There was also certainly some growing pain and apprehension at the individual developer level as they became concerned that projects didn't really have a choice to develop software the way they wanted. That all just contributed to difficulty of adopting the reuse software (and its component-framework implementation) as the baseline approach for future development, despite the goodness of the architecture.

[19] Interview quote, subject #2

3.3 Product platform experience

To address the lack of portability, the thought proposed was "let's build [a more integrated] hardware/software"[20] product as the reusable unit. This idea was intentionally aligned with the *Product Platform* methodology, proposed by Meyer and Lehnerd[21] and enjoying popularity at that time (2002). Though the core definition of a product platform is a "set of subsystems and interfaces", the general aim by the reuse program in the next phase after their first 2-3 clients was to provide a collection of software (possibly a subsystem) that would be most easily adopted by a client project. It was also noted that although first client projects saw a significant reduction in code development effort (a major success for the reuse program), the integration and test phases of the projects were still costly.

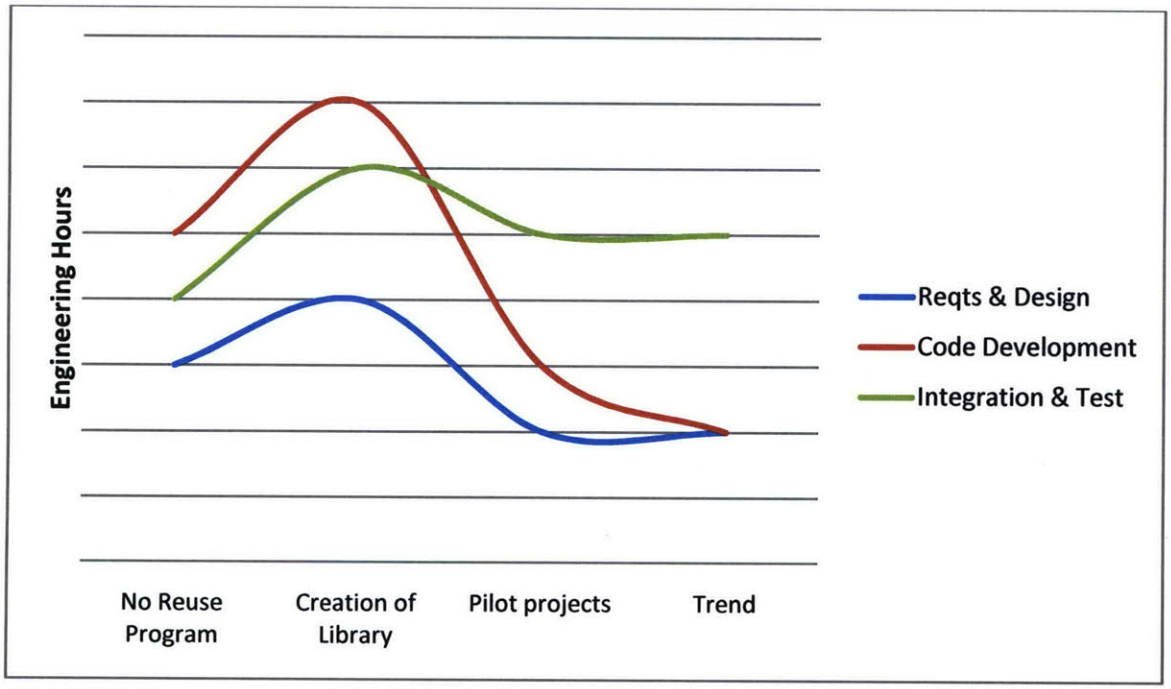


Figure 5: Code development reduced, but Integration and Test Remained Steady

The organization's technical leadership was proud of the reduction in code hours, but felt that the overall savings was not enough to prove that the investment in the reuse program was making the business immediately more cost competitive. At this point, the

[20] Interview quote, subject #2
[21] Mark Meyer book and article on SW PPs

product platform idea seemed to be a logical next step. Another contributing factor was that 2-3 upcoming new business opportunities had significant alignment in functional requirements. The group conceived of a possibility to "design the waterfront of what all customers would want."[22] Most of that design would be the product platform.

This approach would also serve the purpose of getting projects up and running faster. The nucleus of the product platform from the software perspective (there was also a hardware component) was a combination of the framework and components from the library into a few common 'threads' of processing that were reusable as a group. These threads would perform major functions and could be further combined to tailor the product for the nuances of the individual project. In this way, the product platform was in tight alignment with their inspiring vision, summarized below from (Meyer '98):

> "For a software company, the benefits of a robust platform include the ability to more rapidly and flexibly create products tailored for particular niches within the company's market focus. This is achieved by building small, solution specific modules that plug into a larger, more generalized foundation of common program code that provides base-line functionality for all specific solutions. This approach can lead to market advantage through more timely new product introduction and a richer product family..."

Creating a product platform required still further investment. Fortunately, two or three years had passed since the initial investment of IRaD (Internal Research and Development) funding and the company was willing to swallow the next pill of investment in a more vertically integrated product. The product platform would build on the increasingly mature reuse library of its first investments and become a system unto itself. By system, we mean an interconnected set of components, running on its own hardware, that accepts inputs and performs a function of value as an output. In keeping with the product platform model, the system, or "common product" as it became known, would have some common components and threads that were used in most versions of the derivative products. It would also have customizable or extendable portions that enabled the common product to meet specific and larger sets of requirements than those met by the common portions alone.

---

[22] Interview quote, Subject #2.

We will now examine some of the pertinent aspects of creating and executing a product platform.

3.3.1 Project Team Organization

The organization of the reuse program and product platform team evolved to support the changing needs imposed by the business area. In addition to the 'client project' model discussed above, they created a project approach called a "model" where portions of the project that overlapped with common product scope would be performed directly in the reuse organization. Models were supposed to enable the newly developed software to be as 'common' and ready for reuse as possible. We will also refer to this type of team as the 'common product' team.

Though models showed some success (as measured by downstream reuse of their product), the multiple constraints within the business area caused the creation of another project type that we will call a "hybrid". A hybrid project used a mix of developers from the common product organization and those from the client project areas. Hybrid projects were located in a physical space away from the common product group. The driving constraints for these types of projects were: security (customer classification dictated the project be behind closed doors), a strong customer team identity, and freedom to develop a system that was not intended to be common.

The type of team organization chosen by the business area for each new project is one of the important decisions that have significant influence on the execution of the project. A model type project will yield a body of software that is more easily reused, but the team will be pushed and pulled in many different directions. Due to their charter (of developing reusable code), their requirements set will explicitly come from any project that is expected to use their software later. They also do their development in more of a 'fish bowl' where managers and senior developers that might be using their output in the coming months will spend effort (and distract the team) trying to impart their influence.

They hybrid team has fewer of these distractions, especially if they are separated from some outside influences by their location in a classified area. They have the benefit of experience as part of the core reuse team (at least part of the team comes from the

common products group), and try to apply that to the project at hand. The downside is that they are located with, and concentrate on, a project that usually has a tight deadline and a narrow set of requirements. Overall, the sponsor's experience with reuse and product platforms included examples of all three project organization types with varying degrees of success. Some lessons can be learned, but the choice of project type is just one of the pertinent factors to be considered when designing a project.

3.3.2 Challenges:

Requirements: As mentioned earlier, at the time of the product platform's creation, the organization saw critical mass of new business opportunities gathering. The intersection of the requirements from each of the new pursuits was easily identified as being part of the common product's requirements. The reuse group quickly realized that between the three potential projects, there was some variation in the completeness and imperativeness of their requirements sets. The project with the most concrete specification had more influence over the requirements to be levied on the common product. Similar situations would arise at many other times as the common product would always be an object of influence. Development teams saw value in being most closely aligned with the product platform as it offered more chance for internal investment funds, and more opportunity for visibility and possibly growth with customers.

Funding: A major challenge facing the software reuse program and the product platform (by extension) was its funding model and sources. Despite the common knowledge and agreement that the reuse program would require internal funding to create the software library, and later the common product, there was disagreement as to what the ongoing business model would look like. These relative quantities can be seen below in Figure 6. Two aspects of the business model were clear:

- Creation of the repository and initial product platform implementation would be internally funded.
- Customer contracts would be secured for projects taking advantage of the common products. These contracts were expected to be of significant value to justify the internal investment.

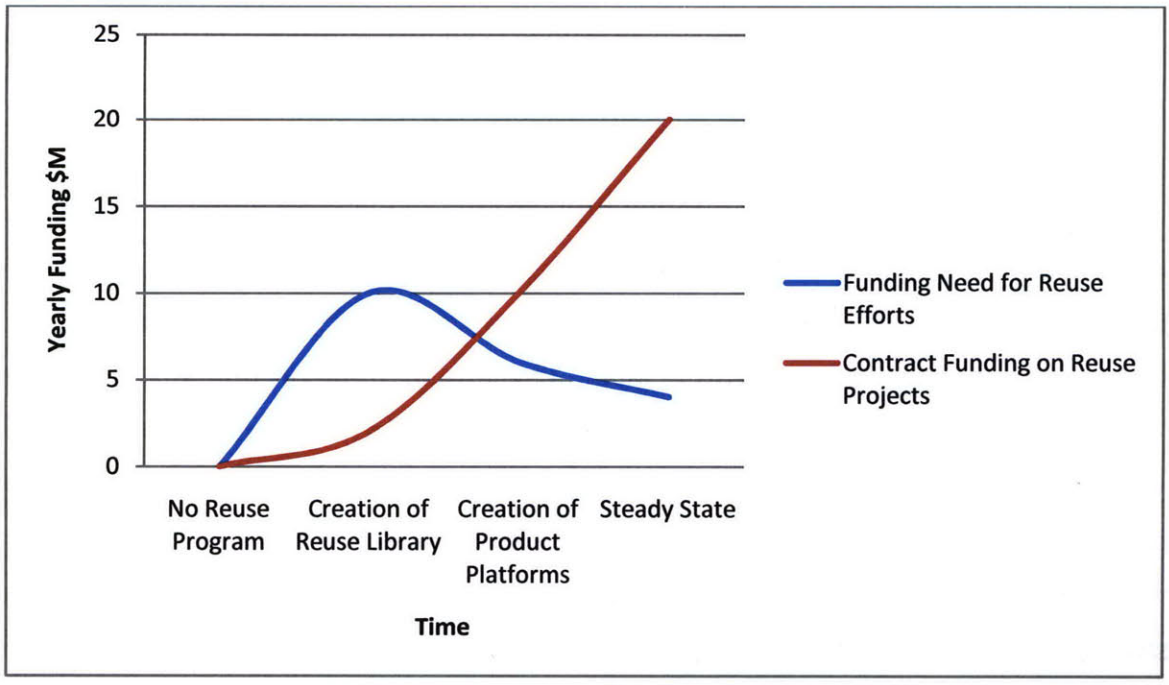


Figure 6: Reuse project funding

These concepts are consistent with many views of software reuse including Jacobson et al '97 and Poulin et al '93. The challenge came when the library and product had been established and new contracts were enjoying the benefits while moving the feature list forward. But who paid for maintenance?

[23]

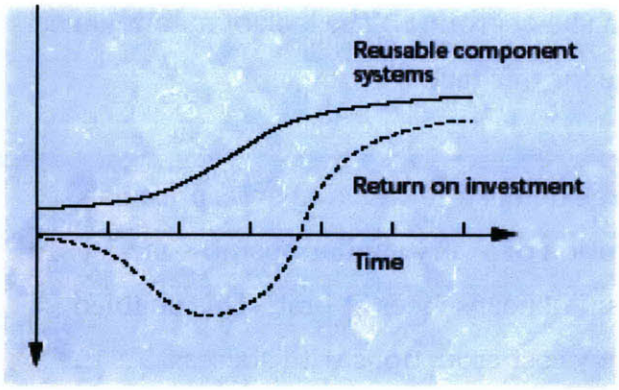


Figure 7: Return on Investment takes time

Three sources of funding were at play in this situation: investment (R&D), contract, and operational. The investment was made, the contracts were kicking in, but the plan for the operational funding was thin. Four possibilities were debated:

Option 1: Pull operational funding from other portions of the business.

[23] Jacobson '97

Option 2: Charge license fees on contracts (at a level sufficient) to support maintenance

Option 3: Pull funds out of each contract to explicitly cover maintenance.

Option 4: Divert some operational funds from the business unit.

Option 2 was selected for its straightforward nature and avoidance of a long hard road of debate over 'rice bowls' that might come with options 1 and 4. Unfortunately, this decision was not without complicating facts. The launch of the product platform coincided with arguably the height of the competitive environment in the last 20 years. The client had only won three projects since the initial investment in the reuse program, and one of the pursuits that helped shape the common product required an extremely competitive proposal. This caused the end result to be a license levied on the potential customer for that contract, but at a price of $0. This set the precedent of maintenance being essentially unfunded from the outset. Additional R&D funds were arranged for the first couple years, but this maintenance task did not meet the client's definition of R&D, so it was not to last. Without giving away the punch line, it should be disclosed that the reuse program as of this writing is on shaky ground. The lack of maintenance and tech refresh funding is not without blame for this fact.

In contrast, a distant portion of the same large defense contracting enterprise has developed a reuse program with better execution of a very similar business model. In their case, 'Option 2' was also chosen, with small license fees at first. This enabled customers to adopt the reuse library and enjoy cost reductions with success. As the amount of projects reached critical mass, the license fees were raised to meet the maintenance need. The level of impact on the project is only enough to cover the cost of maintenance, and so it is accepted by customers.

Customization: The next challenge facing the reuse program was the amount of customization to allow for each project. The client knew of some data points in the customization debate:

- ESU Technologies: This tier 3 supplier of RF and digital receivers had latched on to the right niche some years earlier and had such a large customer and deployed product base that it could afford to turn down requirements to customize its product. Any customization desired would be paid for directly by a contract, and that solution would not necessarily be supported in the future unless sufficient quantities were ordered. This stance by ESU was a luxury unique in the industry.
- Black and Decker and The Honda Element: These product platform trade studies demonstrated textbook application of common platforms with customization for individual applications. The major distinction was the commercial market they served. B&D and Honda had the latitude to design the product and implement the customizations as they wished, simply taking the risk that sales would come from a well executed product.
- Typical defense contracted electronics system: A defense acquisition customer assembles a contract oversight team. The team of program managers, technical 'experts', and logisticians have the most contact with the contractor. They develop or approve the requirements specifications, design, and many aspects of implementation. The contractor does not have the latitude to develop the product with little customization or it will not meet the customer's unique requirements. The customer can demand customization because "he's paying the bill".

Although the sponsor reuse organization strived for example #2 (B&D/Honda), they were pushed by project contracting oversight staff into the typical customized solution described in example #3.

Determining the extent of the common product: Product platform theory distinguishes between the actual 'product platform' and 'derivative products'. The product platform is what's common to all applications, while the derivative product can be derived from the common portions or extended/customized. Originally, the sponsor's reuse organization limited the scope of their product platform to:

- Some hardware (that which is not platform or customer-specific)

- The basic reuse library made up of threads in the component-framework architecture.
- Some software services like startup, shutdown, and tasking

This scope helped to contain the cost of the first iteration of the product platform. A DSM (Design Structure Matrix) showing this structure is below in Figure 8. In a DSM, long stripes of vertical entries are consistent with a module, or module(s), that interacts with many other modules, like a communications bus. The vertical grouping along the left is definitely a bus, and by examining the file list that created this DSM, we know it is the framework. This portion of the system performs all of the command, control, and communications, and was part of the common product platform. The services portion is also a module that interacts with many others. The interesting observation is that the other busses in this architecture were not part of the original product platform.

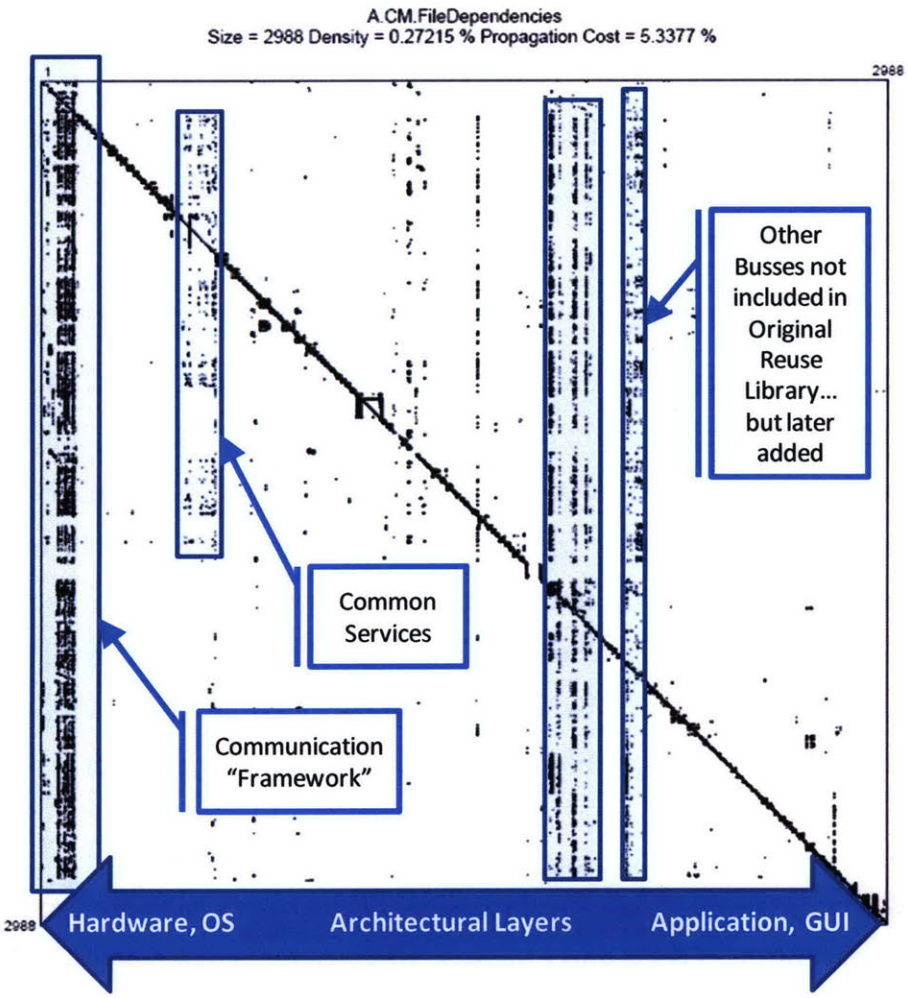


Figure 8: DSM showing only some vertical 'busses' were common

Following the Mirror Hypothesis[24], the product platform *organization* also did not originally include project staff responsible for those other busses not included in the common product. This would prove a difficult test of the architecture, the organization, and the implementation of the product platform idea. The first couple projects executed development through the basic threads relatively smoothly. Progress metrics showed development on track and early indicators supported the reuse program as a success. The next steps after completion of the common threads were for the client projects to develop some of the higher level portions of the software system. Those were to include the busses not covered by the reuse program; indicated by the vertical bars in the right hand side of Figure 8. The client projects had some difficulty with these layers of

[24] MacCormack, A. Rusnak, J, Baldwin, C. (2008). Exploring the Duality between Product and Organizational Architectures: A Test of the Mirroring Hypothesis. *Working Papers, v. 5.*

processing and logic, namely higher level tasking and the graphical user interfaces. The Odin team had excluded these parts of the system on the basis that they were too close to the user's preferences and specific operating concept for each client program. The reuse program was supposed to focus on the common threads of a typical subsystem, and not customize too much.

Despite some difficulty in communicating interfaces from the core reuse group to the client projects, the projects completed their work on the tasking and GUI software and declared success. A pivotal moment came when the next similar set of client projects came along and tried to reuse these two higher level layers of software. The new reusers found that the higher layers had to match their use cases to be reused directly, but the match was less than perfect, so major modifications must be made. In this situation, the two project groups are in opposition about which implementation should be part of the common product. A decision can be made for one or the other, or to support both, but in any case, it can be a painful problem to solve and leaves the product less robust due to divergent baselines.

Consuming and returning: If we mark the start of the software reuse process when a developer grabs a chunk of code and reuses it, then we mark the transition to 'challenging' when that developer tries to return or merge his modified code back to repository from whence it came. The real issue becomes how to support the first stage of 'clone and own' and then later the need to return improvements, bug fixes, and enhancements to a 'common baseline'. The product line approach sells itself based on the ability to take updates that one project makes and 'seamlessly' apply them to other projects in the product line. The need exists to enable simultaneous development of the same software, branching, and uncommonality. These issues seem minor when we talk about one source code file, but they become almost intractable when the baseline to be merged is thousands of files. Starting reuse and branching is easy, trying to merge and keep all the reuse clients satisfied is hard.

3.3.3 Product Platform Architecture

The software architecture defines the structure, components, and interfaces of the system. From Jacobson et al '97:

"Getting the architecture right provides many advantages. It ensures that many organization units, scattered in space and time, can work on different components and rest assured that they will come together successfully. It ensures that a system put together in this manner will have integrity and a robust structure. It enables maintainers later to modify local parts of it without disrupting other parts. This ability to modify the system successfully enables it to evolve, as its environment changes. In turn, successful evolution extends the life of the system.[25]"

The sponsor's component framework architecture was one of the overall strongest successes of the reuse program, and certainly contributed to its longevity. It helped achieve almost all of the advantages cited above. The DSM shown earlier indicates the modularity of the architecture. The anecdotal input on how hard the development team worked to stump themselves during development supports the fact that it was a well designed, extensible architecture.

3.4 Case studies

A selection of brief case studies will be presented below to highlight the sponsor's varied experience with reuse and product platforms. A pictorial timeline of the projects is shown below in Figure 9.

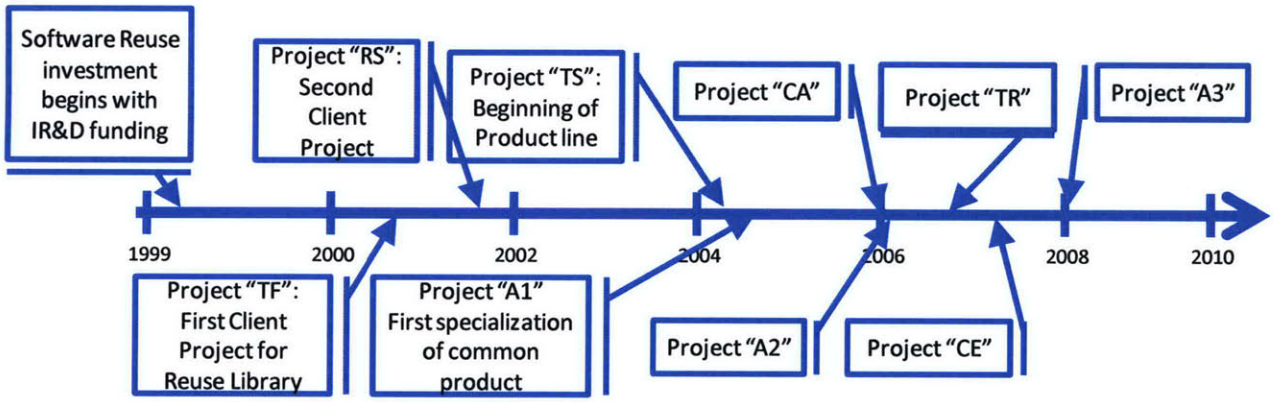


Figure 9: software reuse and product platform timeline

Case #1: Project RS – This project was the second client reuse effort after the initial launch of the program. It was conceived as a subsystem upgrade to an existing system

[25] Jacobson et al., '97

that the sponsor had delivered years earlier and was fielded throughout the US Navy. The upgrade consisted mostly of processing hardware and software, and it was very similar to the original model of systems that the reuse program team had in mind when they created the component framework architecture. These two facts made it a good candidate for an early adoption project. The development sharing model for this project was that of a strict 'client project'.

In this model, the reuse library was delivered in releases to the project. The team was located in a different location than the core reuse developers. The project team was expected to perform almost all of the development by themselves using release documentation and only consulting with the engineers that created the reuse library. If problems were discovered by the project team that required a change to the core library, those problems were supposed to be fixed by the core team. This created some tension between the two groups when the project team felt they could not move forward until a problem with the core library was addressed. Tension was compounded by the fact that the reuse library was still immature at this point, having only a few demonstrations and two projects exercise the code before project RS had to 'make it work' for a real customer with budget and schedule constraints. The customer was, of course, sold on the merits of the reuse library as part of their decision to award the program to the client.

This project was considered successful by some participants and observers because the technical objectives were achieved, but this came at a cost of 17% over budget and 3-4 months of schedule delay.

Case #2: Project TS – This project is included in this study primarily because it served as the launch of the product platform. It was a moderate sized System Development and Demonstration (SDD) project. This type of project is expected to be the transition from early phase, technology driven endeavors (that may not meet all the requirements) to a mature system with proven readiness to be used by DoD warfighters.

Project TS just happened to be executed around the time of the reuse program's transition from a simple software reuse organization to a product platform group. Some

would say that the TS project was also a must-win for the business area, and it was proposed against a very competitive landscape. It could be further speculated that the only way the team could propose a price that was competitive enough to win would have been with an existing product baseline that the business invested heavily in. Because of this, the TS project's influence on the transition to a product platform, and the implementation of that product platform, was strong.

The challenges seen by this project included two from the list in section 3.3.2. First, the requirements set for this project was large enough that the collection was seen as a 'superset' of the requirements one might expect for multiple smaller typical systems. This drove the architecture team to generate a system specification for the product platform and for the project that was lengthy and detailed. It described a system that, if implemented, could meet the business area's needs for reuse and follow-on projects for many years to come. Although this was a feat of systems engineering, it challenged the project and the reuse program to determine what to implement on the current funding line and what to defer as nice-to-have, but unnecessary growth items.

The second challenge that took major influence on this project was the fit of the fledgling product platform to the business' cost model in a way that would generate a sufficient funding source to enable development. Although significant IRaD investment had already been devoted to the reuse program, still more was needed to create the product platform. After this hurdle was cleared, the real issue became finding funding for important changes to the baseline that were not directly tied to project requirements. In this case, a common product program should have been able to fund these common maintenance type issues. The project was proposed with the license fee option, but by the time it was on contract, the fee had been reduced to $0.

Case #3: Project A1 – This project demonstrated how the issue of customization was a needed part of the strategy for a product platform. The amount of customization each project required would need to be considered for its impact on the product platform. At this point, we should discuss a term that we defined, but haven't used yet – product line. The best case is if a new project enhances functionality of an older product, and this situation is even better if the new product is planned to be the next in the product line.

The architecture of the product line should support the expansion of capabilities, and the iterations that implement them. "Getting the architecture right successfully enables [the system] to evolve... and evolution, in turn, extends the life of the system.[26]"

Project A1 was a quick turn-around effort that required iteration #2 of the product platform to be designed, implemented, and tested within a year. This schedule pressure also made the project unique. The understanding by the engineering team was that the customer needed the system (which was an upgrade to an existing system supplied by another contractor) to be deploy-able within a year. In concert with the theme of the product platform, this system would be the next iteration of the common product, based on project TS (the first iteration), but a 12 month delivery would make that challenging to say the least.

At the beginning of the project, the engineering baseline was to completely reuse the core system (up to, but not including the Graphical User Interface, GUI) from project TS, but put a new presentation layer on it by changing the GUI only. The GUI changes would be necessary because the customer's plan to operate the system was *slightly* different than that of the TS customer. The degree of the difference was an incorrect assumption perpetuated by a lack of completeness in the customer's specification. As the significance of the difference in operating concept (or CONOPs) became fully understood, the product platform team realized, from their systems engineering experience, that changes in CONOP can have deep rippling effects on system design. Figure 10 illustrates the decision point facing the product platform team: to embrace the change in CONOP and the changes in direction it would cause on the common product, or to hold the design firm and try to find a political solution to the problem. This might mean retraining the existing operator base to use the system in a way that was different than the legacy system A1 was replacing. A solution like this is not simple or trivial to 'make happen'.

[26] Jacobson, et al. '97

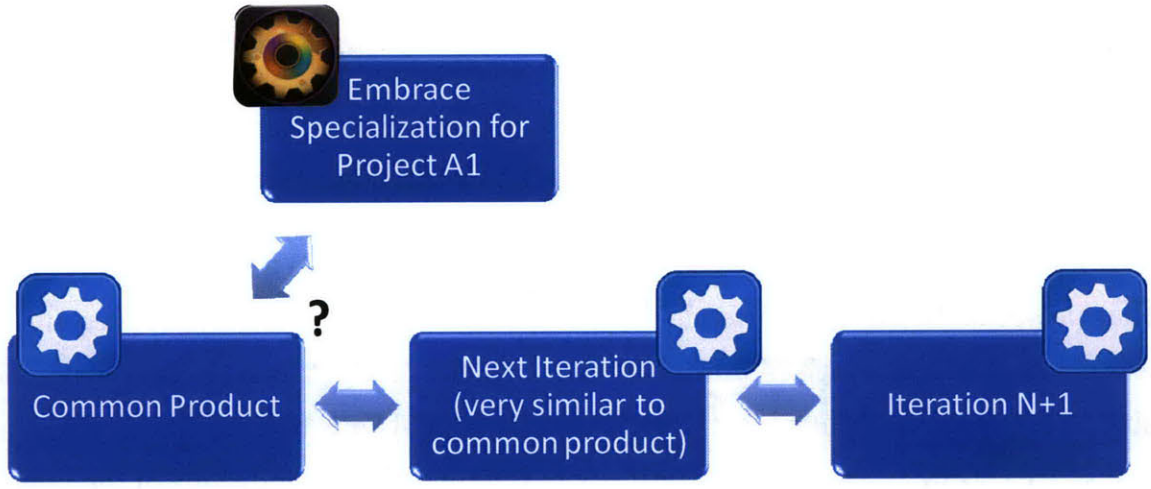


Figure 10: The decision to embrace specialization or not

The management team of the product line made little of this decision. They hardly realized that a decision was upon them, and that fact is key. The reason is that years of custom product development had made them firmly believe that the systems should do what the customer dictates want; after all, they're paying the bill for the engineering time. This lack of strength in direction of the product line was facilitated by the schedule push – the common perception was that a political solution might take months to get full agreement and customer approval. To meet the deadline, the 'work-around' that allowed the changed CONOPs and system design might just be easier. The unforeseen issue was that this would create larger problems down the road when the product platform team tried to maintain a common baseline between the portfolio of projects and systems.

Project A1's change in CONOP resulted in some modest changes in design of the core signal processing of the system. The impact was felt more significantly at the database and GUI layers. This highlights the challenge of determining the correct level for reuse. What parts are in and what are out? If the GUI itself should have been out, then no project would be able to easily reuse the GUIs from A1 if they wanted to use the system in that way. If the GUIs were in, then maybe they should've been better designed for reuse, layering. Maintaining a common GUI is hard; it must be "thin" layer so that the changes in concept of operation don't affect it as much.

Case #4: Project TR – This project was also an illustration of the challenge of accepting a customization request. It is a case where the motivation for that request was very strong – to fill an unmet need in a Quick Reaction Capability manner. The case is detailed in section 4.2 as it also demonstrates the hybrid team example.

We draw valuable insights from these cases and present them in a manner useful by similar projects. Further, experience with the product platform technique presents cases that may reveal how it requires the rigor of strict product focus to best serve the business. The main output of this work is to offer conclusions that can be used to shape business area strategy and reuse techniques based on specific conditions of the potential projects or product families. The product platform approach has served the subject organization rather well, though not without its bumps in the road. As we will discuss in section 5, there are conditions under which the product platform will work best. Many of those, but not all, have been exhibited on the Odin reuse program.

3.5 Preliminary hypotheses

At this point in the research, we can articulate four hypotheses. They are certainly preliminary, though informed by our literature search and interview responses.

1. First, successful reuse programs take time to establish and pay dividends. Immature ones create an environment where client projects are hard to execute. As the reuse library and product platform matures, everything gets easier (socialization, productivity, requirements compliance) and the payback is significant.
2. In a reuse library, it's not too hard to "clone and own", the difficulty comes with maintaining a single software baseline and the merging changes back into the baseline.
3. A successful reuse program requires a funding source for maintenance and tech refreshment. The business model must support this, and it should probably be an operational cost expense rather than research and development investment.
4. Highly leveraged projects create a high stress factor environment that results in lower cost performance and development productivity.

In the next section, we will analyze project data (cost, productivity, reuse %, system performance) and survey results to search for patterns of causality that reinforce or weaken our hypotheses.

4 Analysis of Cases

To this point, we've discussed logical cause and effect relationships that present certain hypotheses about software reuse in the defense electronics industry. These hypotheses are substantiated by past literature and interviews conducted as part of this research. We now seek to further solidify the business & technical reuse model presented in section 1 by supporting it with an analysis of the options and strategies open to managers. The data to be analyzed comes from two sources: subject interviews and project data. The interview results are contained below in section 4.1.

4.1 Interview Results

To draw on the sponsor organization's wealth of experience in the software reuse and product platforms, interviews were conducted with many individuals that had involvement with the reuse projects described above. In total, 25 interviews were conducted with developers, architects, and software and program managers throughout the organization. The responsibility distribution of the interviewees is shown in Figure 11, below.

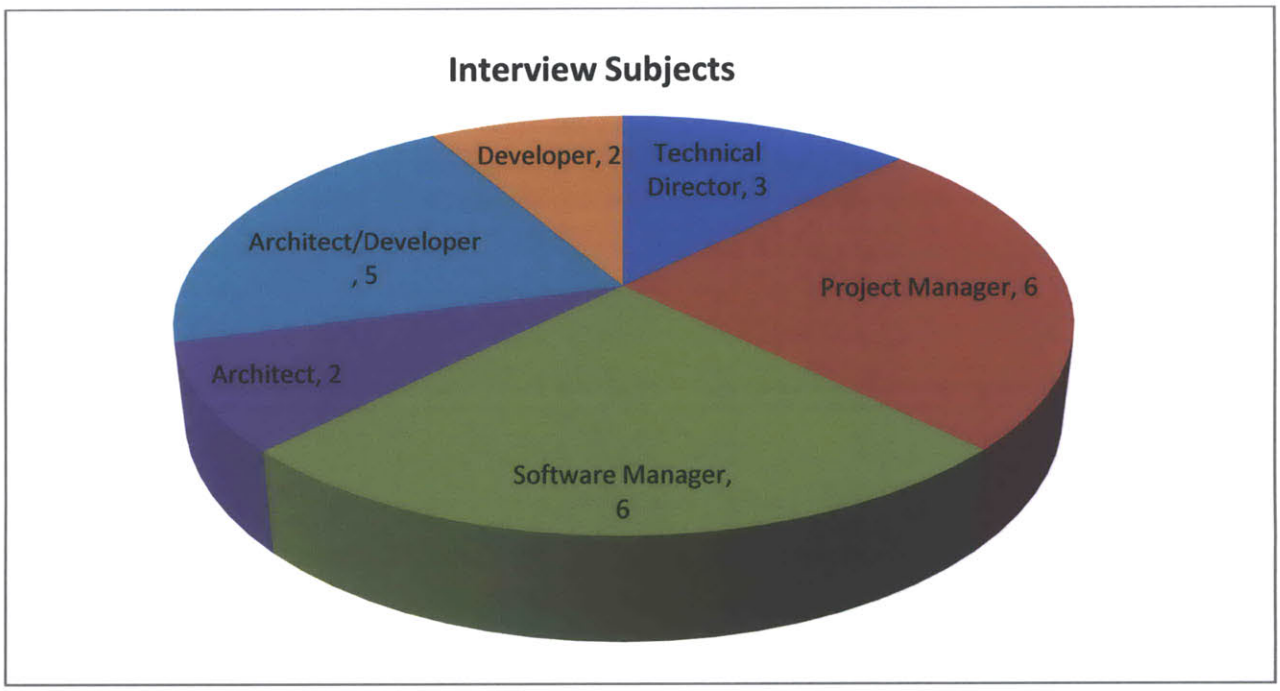


Figure 11: Research Interviews drew from a substantive cross section of the organizations

The respondents represented the gamut of experience and influence in the Odin program, from original architects and developers to software managers and directors on client projects. Drawing input from such a diverse group enabled us to fully represent the issues seen over the lifecycle of the reuse program.

The intent of the interview process was threefold:

1. To help shape the research if subjects had significant insight that could form the basis of a hypothesis.
2. To serve as a data set for correlating patterns within the reuse organization
3. To help baseline the sponsor organization by getting enough views of 'how it really happened' so that we don't rely on a single person's view of past events.

The interview subjects were asked a series of questions in two types. One type was an open-ended question that allowed them to provide as much information and/or opinion as they chose. The other type of question required them to rate some aspect of the project, organization, or system on a scale (1 to 5). This type of question can later be used as a data set. The interview instrument is contained in Appendix 1, but some representative questions are:

> Reuse as a Design Driver: How important was software reuse in the architect and design phases?
> What factors constrained the project significantly?
> How successful was the software reuse initiative on this project?

The results of the interviews were informative. They paint an effective picture of the reuse program within the organization. Some highlights of the interviews are shown in the histogram plots to follow, while the full set of notes and quotes are contained in the appendix. First, we see from Figure 12 that most interviewees chose to rate 'reuse' as an important design driver between 4 and 5 on a scale of 1 to 5. This tells us that the overall reuse movement was important across the organization on many projects and to many individuals.

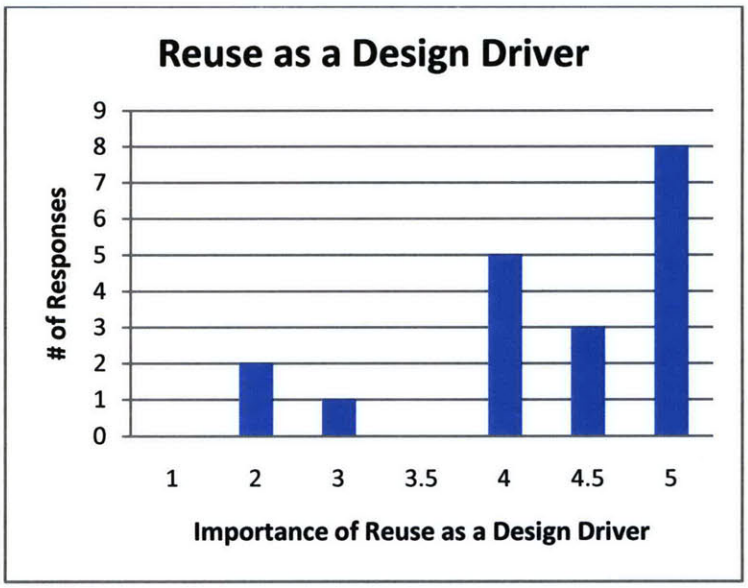


Figure 12: Reuse was often an important design driver

Figure 13 summarizes subjects' responses when asked if the software reuse library was suitable to their needs. This is an umbrella question to basically ask if the developers felt they were being provided all the tools they needed and could get the project executed with the reuse library as an engineering baseline from which to start. Overall, the responses showed that the repository was suitable. This might indicate that the R&D investment needed to create the repository was worthwhile.

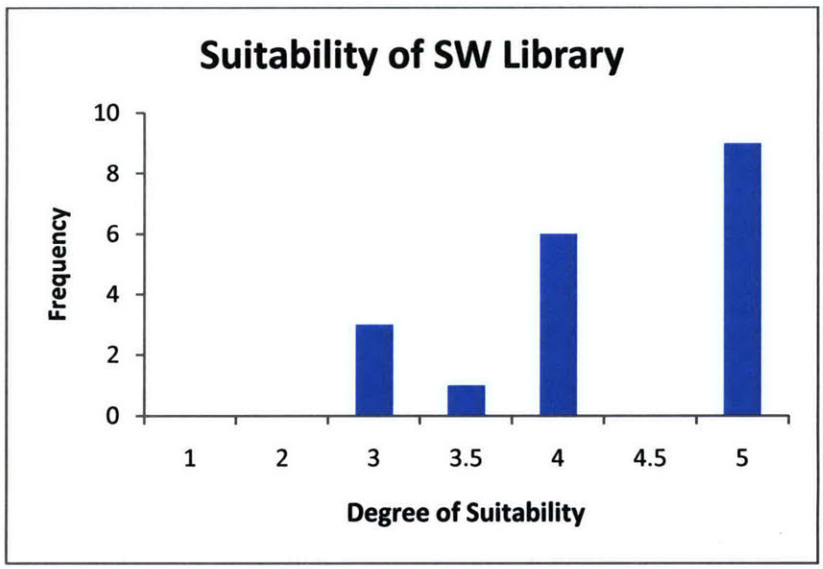


Figure 13: The Software repository was largely seen as suitable

In order to baseline the projects, the subjects were asked if the projects targeted for reuse were technically challenging or not, as compared to others in their career. The responses were quite varied, showing that the organization did not select especially hard or easy projects as reuse candidates.

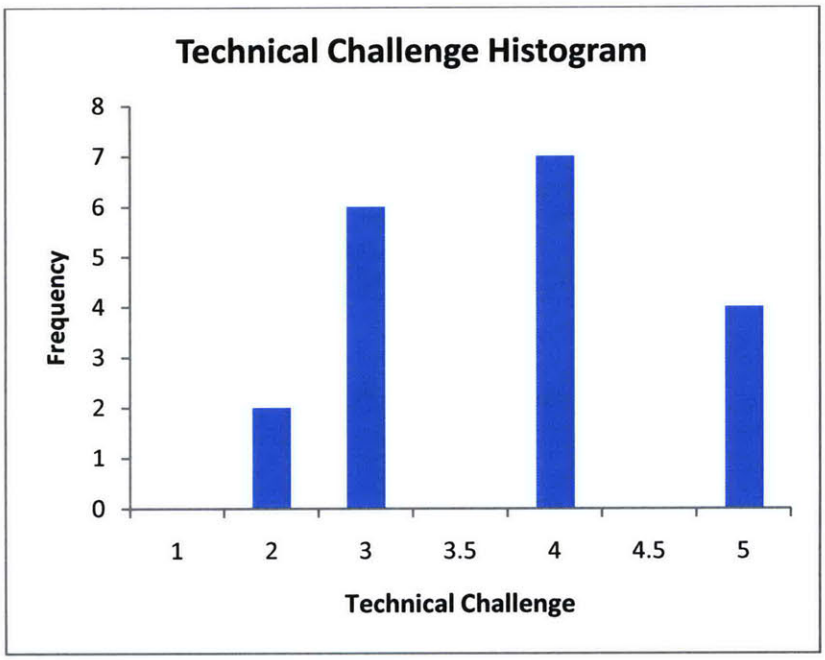


Figure 14: The technical challenge varied as expected

Figure 15 shows responses to the question about how well system performance was achieved. A rating of 4 meant "met expectations", 5 meant "surpassed", and 3 meant "slightly disappointed". This data shows that the subject systems largely met expectations, but some fell short. This variance may be indicative of many factors at work simultaneously. Over the decades of past projects, the sponsor had a reputation for technical excellence but poor cost performance. This data could be explained as being reflective of a more competitive business environment than the years prior. More competition would lead to cost cutting, and system performance might suffer. An alternate explanation would be that the reuse program blunted developers' focus and system performance occasionally suffered.

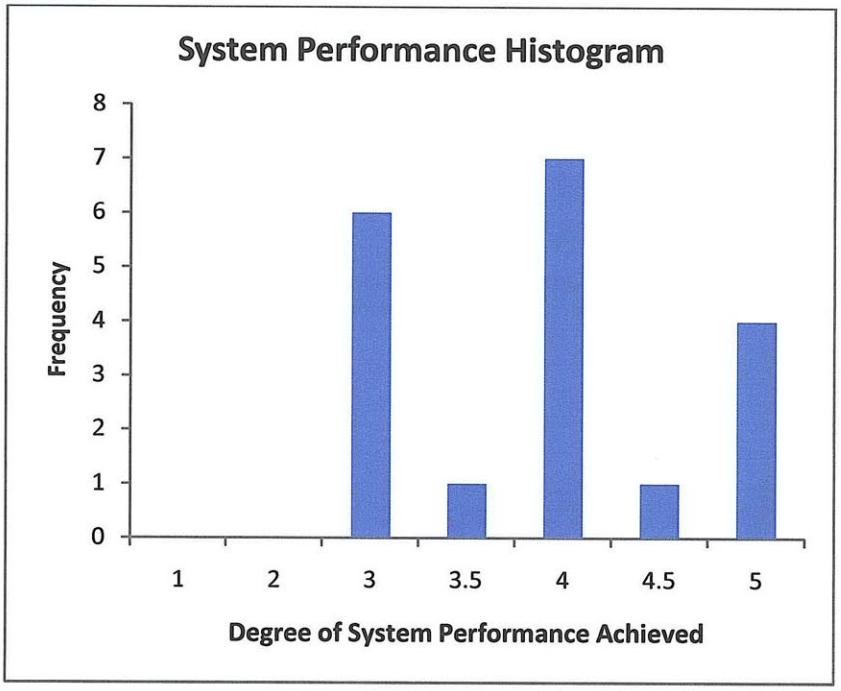


Figure 15: System performance was usually achieved

Figure 16 shows the perceived success of the team when measured by their ability to participate in and contribute to a reuse program. It has the same definitions of success as the previous system performance question. This data is skewed slightly higher than that of the system performance assessment. Besides simply achieving more reuse success than technical performance, an alternate explanation could be that the projects could be seen as successful against this measure by simply using the reuse baseline. Being a software reuse "taker", but not a contributor back to the baseline would be a case of less overall reuse success, but might not reveal itself in this interview. We will consider the achieved reuse success on par with the achieved system performance in this collection of projects.

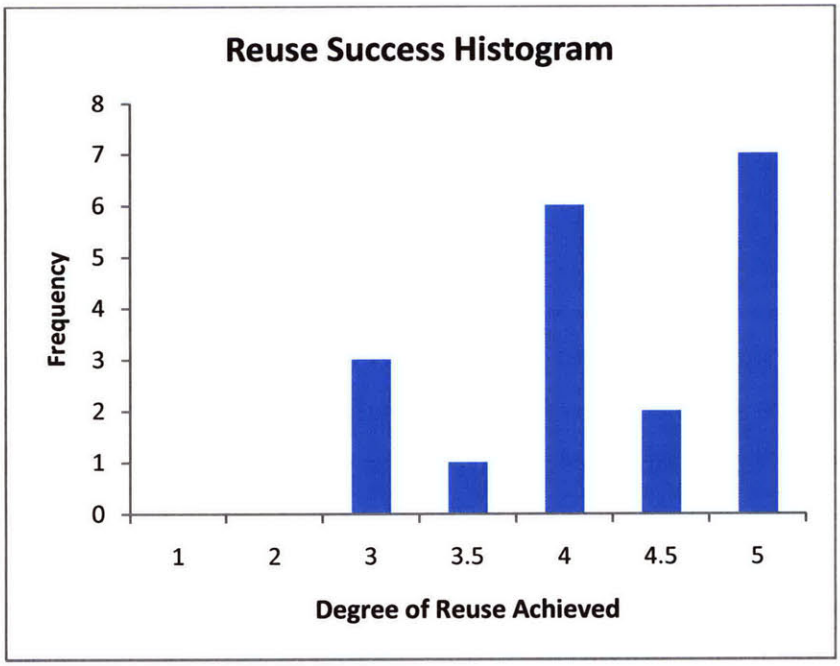


Figure 16: Reuse was also successfully achieved

4.2 Project Variables

We will now examine the cases discussed in section 3.4 by tabulating and correlating the metrics and data about each project. Some of these metrics are secondary values that are only revealed when one looks hard at the reuse patterns of the organization. Although many variables will be used, a summary of some primary reuse dimensions are shown below in Table 1.

Table 1: The management 'variables' present in a project's design

Variable	Metric	Shows
Project Team orientation	Team organization type (common product team, client project, hybrid)	The proximity of a project to the business' core focus
Planned Reuse	Target reuse % (by LOC from SW estimate)	Management direction
Challenge of system	Technical challenge on 1-10 scale	Tension between Design for

performance		performance vs. cost, or reuse
Available SW library	% of components planned for reuse (from SW estimate)	Do the needed components exist already? Is this a big part of the new mainline?

Project Team orientation summarizes the overall approach taken by management as they plan to execute a project and give it the highest likelihood for success. It is made up of project team design and a centrality of the project to its core reuse library or mainstream software development of the product platform.

From experience in the wider industry as well as in the subject organization, there are fundamentally different team organizations for reuse. We will discuss three main types, as they persisted in the sponsor's software development organization. The types, enumerated in Table 2 and Figure 17, are also described in further detail in section 3.3.1.

Table 2: Types of development teams

Team Type	Description
Common product team, or "model"	Integrated with the reuse organization
Hybrid team	Pulled from the reuse organization to cross-pollinate, but physically separated
Client Program	A client user of the software family

Figure 17: Project Teams vary in proximity to the "core"

The other major contributor to reuse strategy is how central the project is to the overall product line. In the client organization, some development efforts were vital enough to the business' success that the organization was expected to morph around that project to make it successful. These projects would have some strategic value such as 'TS' and 'A1' in Table 3, below.

Table 3: Uniqueness can drive a Project's strategic value

Project ID	Reason for importance	Description
TS	Core Product	This project had a 'superset' of requirements and could form the basis for reuse by many related systems. It would be the flagship of the product line.
A1	First Product Platform 'Specialization'	This project was the first to reuse the core product and specialize portions of it. It was the first example of true software reuse as it was intended in this product line.
TR	Client Program	A client user of the software family

The sponsor business executed subject projects over roughly 10 years in software reuse effort. These projects are shown on a timeline in Figure 18, below.

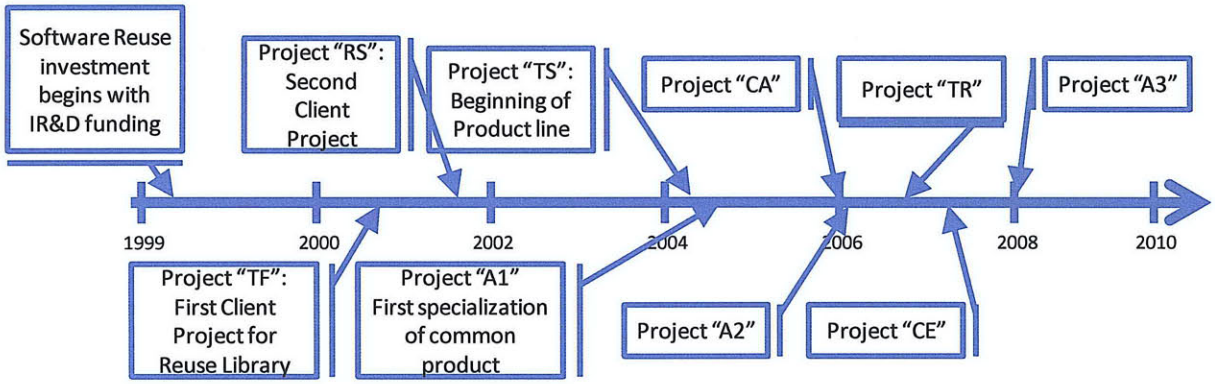


Figure 18: Software reuse project timeline

Project TR is an example of another point in the 'Reuse Strategy' space. This project was characterized by an urgent need to satisfy the customer's schedule requirement. As a defense contractor, some of the business' proudest moments come from meeting the warfighter's urgent need as fast as possible if it can save the lives of our soldiers. The TR project had such a need to bring a system to deployable maturity quickly and fill an unmet requirement.

The system created for the TR project was also different than earlier instantiations of the product line. It had some new capabilities and lacked some existing ones that gave it and an unclear future in the line and allowed some principals to debate its place in the reuse organization. What followed was a rationale allowing the TR project to be physically separated from the reuse organization. This was motivated by the thought that the system could most expediently be developed away from the distractions of the larger organization. The resultant system (with its strange mix of capabilities) was also expected to be more of a 'one-off' and could possibly not need to be returned to the baseline.

Planned Reuse is the extent to which the project expects to reuse existing code from the software library. On the surface, this indicates the similarity of the new system to ones already existing in the library. At a deeper level, it might also indicate the aggressiveness that the architecture and business capture team pursued the opportunity. There were cases in the client organization where a system could be best build with method "B" of a set of components, but method "A" already existed in the

product line. Despite lower system performance, method A might be chosen in order to contain cost. A key learning from the planning and early execution stages of reuse projects is that the proposed architecture must be communicated to the development team so that they can build the system by method A, which is the only way they will meet the requirements within the funding available. This planned reuse variable can strongly direct a project's execution and outcome, possibly along important dimensions other than cost, though cost might have driven the reuse target.

The Challenge of System Performance summarizes the influence on the project that comes from purely solving a 'hard problem'. Although Jacobson ('97) points out that a common complaint is that "Reused code components are too slow."[27] The retort of a well-designed code repository should be that "[poor performance] is a design flaw," and performance requirements are also a part of the reuse program. Nonetheless, one could see that when all other conditions are equal, a project with a very challenging technical problem might have a harder time meeting its requirements when directed to use a particular reuse library, even if that library was thought to be capable of meeting said requirements. On the other hand, it is conceivable that a reuse library could include a set of components that uniquely met a common performance requirement, and enabled the system to meet the challenge of system performance.

In the case of the client's experience, very few components or threads have the unique and designed-in ability to meet performance requirements. This means that the challenge of system performance becomes an influence on overall project success. At this point, we should also note that other commercial processes have been considered. For example, an open source approach was examined for its associated modularity that has been shown to enhance reuse. While this is true, it is challenged to meet unique performance requirements with such a modular architecture. Another note is that the sponsor's component framework architecture, we will see, is actually very modular and may not see large benefits in this area from an open source approach adopted primarily for its modularity.

---

[27] Jacobson et al. '97

The Available Software Library and the suitability thereof is another major variable that modifies the potential outcome of a development project whose baseline includes major reuse. The 'design' of a project must be driven by what is available in the software library. Fortunately, this is often true in practice. Only two of the client case studies exhibit poor understanding of the software repository at the project's proposal and inception. In one case (project RS), the reuse repository was thought to be more mature and capable than it was in fact. In another case (CA), the library might have been well understood at the time that the technical baseline was established (prior to actual execution), but the architecture and design teams did not develop the system according to the proposed technical baseline.

In general, if the software library is rich in components that satisfy the project's functionality and performance requirements, then not only is the overall development effort low, but so is the execution risk. The maturity of the library is also a key factor in this area, for if the library is full, but only of 'demo quality' code, the suitability of that library to a major development project with hard requirements will be poor. When this suitability is poor, the project always suffers. It can be minimal, if the gap between understanding and reality is small; or there can also be major suffering, if that gap is large.

4.3 Analysis of variables

Armed with these variables for design of software projects, we can analyze existing project data for patterns that correlate with project and reuse program success. There is much to learn by analyzing past reuse projects in detail. In addition to the 'soft data' approach of the interview, a 'hard data' analysis is fruitful for its objectivity and unambiguousness.

Revisit the research question – From the first general question of:

> What insight does the client's experience with software reuse projects reveal about the impact of input conditions and management decisions on project success?

... we use our project data to search for patterns that lead to the relevant themes of good software reuse. The available project data comes from 3 sources:

Software management metrics: All projects in a CMMI level 3 organization[28] maintain metrics on software development. CMMI (Capability Maturity Model Integrated) is a multi-function process improvement approach and assessment by the Software Engineering Institute. Fortunately, the client organization maintains a SEI CMMI level 3 process in its development groups, and a CMM level 4 process for software engineering. Consistent with a process maturity of this high level is a practice of metric collection. For any project of more than 1000 development hours, software project leadership records at least the data set described below in Table 4.

Table 4: Software Metrics relevant to research

Metric	Shows	Relevance to research
Delivered Lines of Code (DLOC)	Amount of software in full system, as delivered. Includes reused code.	Planned and actual quantities are relevant, and comparison
Equivalent Lines of Code (ELOC)	Amount of software written new or modified on a project.	Will consider planned and actual
Planned SW hours	Planned size of software effort	Represents the size of the project
Actual SW hours	How many hours a project actually took to execute	Represents the size of the project, comparison to plan
CPI, Cost Performance Index [0-1 scale]	How much money an effort spent to execute, relative to plan	A key indicator of project execution
SPI, Schedule Performance Index	How much time it took to execute an effort, relative to plan	A key indicator of project execution
Reuse %	Percentage of delivered software that was from a reuse source (not new or modified)	A central measure of how much reuse a project attained; Will consider planned and actual
Productivity [LOC/day]	How much software is written per unit time (only new or modified)	A key indicator of project efficiency

The most common and important values are: planned code size (ELOC and DLOC), planned SW effort, code development progress and growth, and productivity. This rich

[28] http://www.sei.cmu.edu/cmmi/

data set shows us project trends over time, and actual vs. planned performance of the software effort on a project. The core set of metrics is collected by the client software managers in virtually all projects. Due to the latitude afforded a manager, there are some variances in the full set of metrics maintained when we look across many projects. Some projects may be very small or very short schedule, in which the value of the metric at anything other than a summary level is not worth the time and disruption required to measure it. Despite these variances, the data set is invaluable to this analysis and supplies us with many of the quantities discussed below.

Project Cost Data: Without exception, costs are tracked for every project conducted in the client's organization. This data provides us with an overall measure of cost and schedule performance. It can be examined at any level that the project tracks costs, (at least summary, engineering, Integrated Product Team, and functional) consistent with the project's Work Breakdown Structure (WBS).

Interview Data: The considered viewpoint taken by the engineers and developers that support a project is important as it can incorporate complex relationships between issues that may not be reflected by hard data. In addition to answers to interview questions, some composite quantities and secondary values are used in the analysis. They are listed below in Table 5.

Table 5: Additional metrics drawn from the interview data set

Metric	Shows	Relevance to research
Reuse as a driving constraint (mandated)	Degree to which management levied reuse on the project as a constraint	How much software reuse was 'designed in' to a project
Stress factor	A composite measure of cost and schedule pressure	Illustrates how projects execute in different stress environments
Suitability of Software Library	The development teams' assessment of how well the library met their needs	Represents the readiness of the repository for the project
System Performance	How a system met its performance requirements	Will consider planned and achieved
Achieved Reuse	A subjective measure of how well the	Resultant reuse relative to

(Interview Estimate)	team did at reusing SW, relative to expectations	expectation

Through interviews and anecdotal research, we have developed the following hypotheses about the pattern of project performance in the client's software development environment:

- Without reasonably low expectations for productivity and cost competitiveness on the first reuse projects, those projects will suffer unmet expectations due to an immature reuse product and process. Effective software reuse is only possible once the product and the organization get further along the curve in Figure 19, below.
- High stress factor projects will have poor performance.
- Unrealistic expectations and mandated reuse strategies will negatively impact the development team. This could manifest in lower productivity, system performance, or another quality not captured by the interviews.

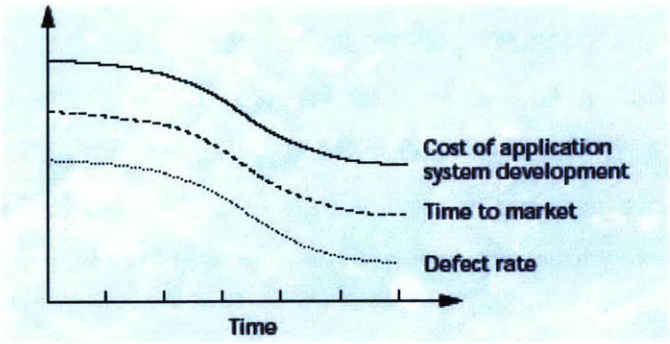


Figure 19: Cost and Effort start high but fall over time (Jacobson et al '97)

The first hypothesis is upheld by our project data, as shown in Figure 20, showing poor cost performance when projects were executed with an unsuitable repository. The converse is also true; a highly suitable repository yields good cost performance.

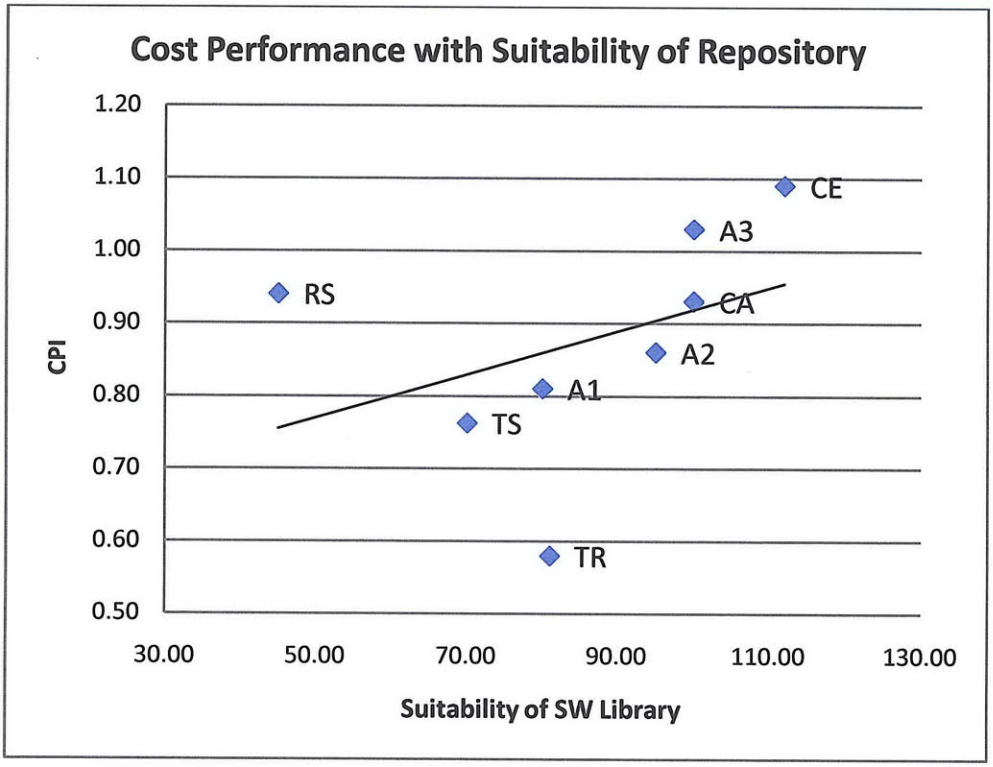


Figure 20: It takes a suitable (mature) SW library to be cost effective

This figure uses cost performance index (CPI) because it is a measure of performance against expectation (actual cost of work performed divided by the budgeted cost of work planned – see appendix). Later patterns will also look at productivity. The 'suitability" quantity is a weighted average of interview data with the planned reuse baseline. It is also normalized by interview respondents' own tendencies towards software reuse.

In this collection of data, we can infer that a significant cause of project failure was an immature reuse product and process. The cases TS, RS, and A3 were all very early in the software reuse program, and consequently early in the lifecycle of the product family. The project team spent too much time addressing shortfalls in the capabilities and robustness of the library and not enough time on the new and planned parts of the project that would get it 'completed' on time. This drove costs higher relative to the plan, which in turn reduces CPI.

The suitability of the software library also has a direct effect on how much code is reused by the development teams. This is intuitive – if the code is worth using, they'll use it –

and serves as a good logic check on this research. Figure 21, below, shows this effect by plotting reuse percentages with the suitability of our case study projects.

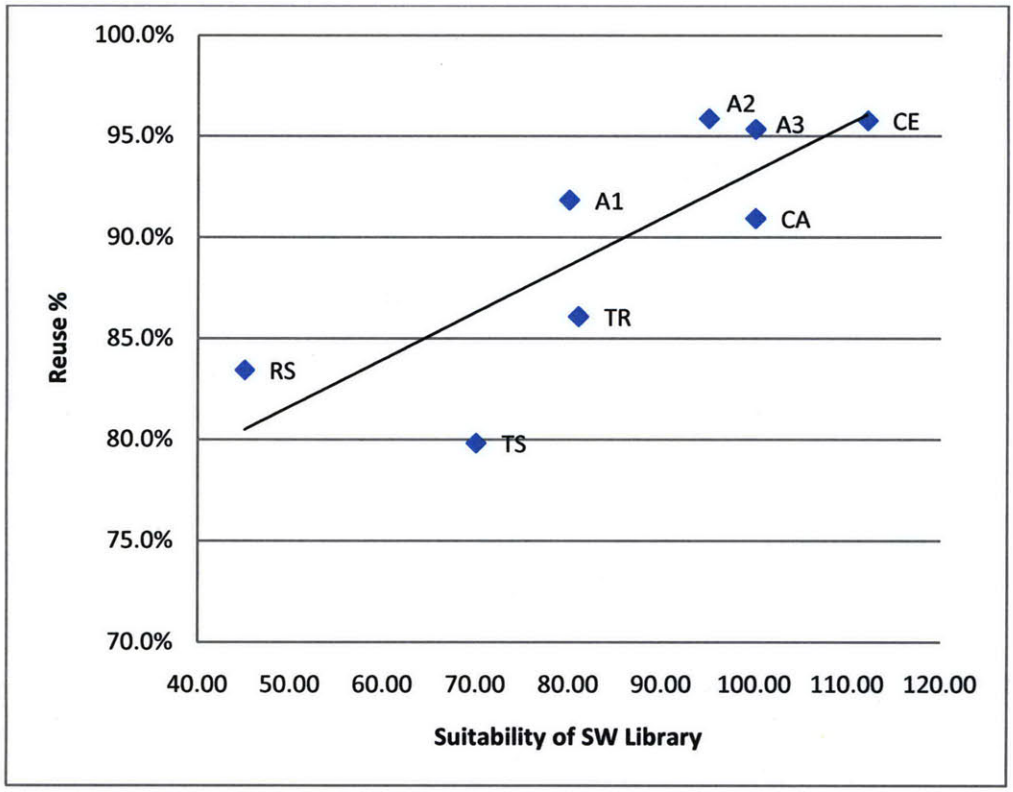


Figure 21: If the library is suitable to the project, developers will (re)use it

Project Organization: Another dimension along which we can compare projects is the type of development team organization chosen by management. For many projects of high visibility, strategic, or reuse importance, management chooses to execute the project in a way that doesn't permit the team much latitude or flexibility. Management's rationale is that they need to 'keep an eye' on that team, providing 'guidance' along the way, and thereby ensuring that the project is a success. See section 3.4, Case studies, for a comparison of the project organizations used by the client.

In Figure 22, below, we see the three project organization types plotted along the x-axis. A centralized, common product group performing the development work is a "1", consistent with Table 2. A team acting as a 'client' of the common product group is a "5", and a hybrid team (somewhere in the middle, see section 3.3.1 and Figure 17) is a "3", or shaded just above or below that value. The numerical values used here for

plotting purposes are measures of the degree of separation from the core reuse group. If the core reuse group executes a project, it is a "1", while a client project is very removed and plotted as a "5".

These organization types are shown with the resulting stress factor observed on the project. *Stress factor* is an averaged value taken from the interview results that indicates the overall schedule or cost pressure of the project. As the pattern and trendline indicates, high stress factors are coincident with centralized common product development teams. This is likely a result of the underlying *cause* of that chosen development team style, i.e. management's desire to closely monitor a highly important strategic project causes the stress factor. This coincident stress factor is useful to managers who might personally find it important to alleviate stress on a project, regardless of its importance. A suggestion might be to employ an opposite strategy in these cases: stack the team with highly capable individuals, and let them go work in an apparently low visibility manner. Next, we will see that this stress factor leads to other negative outcomes on a project.

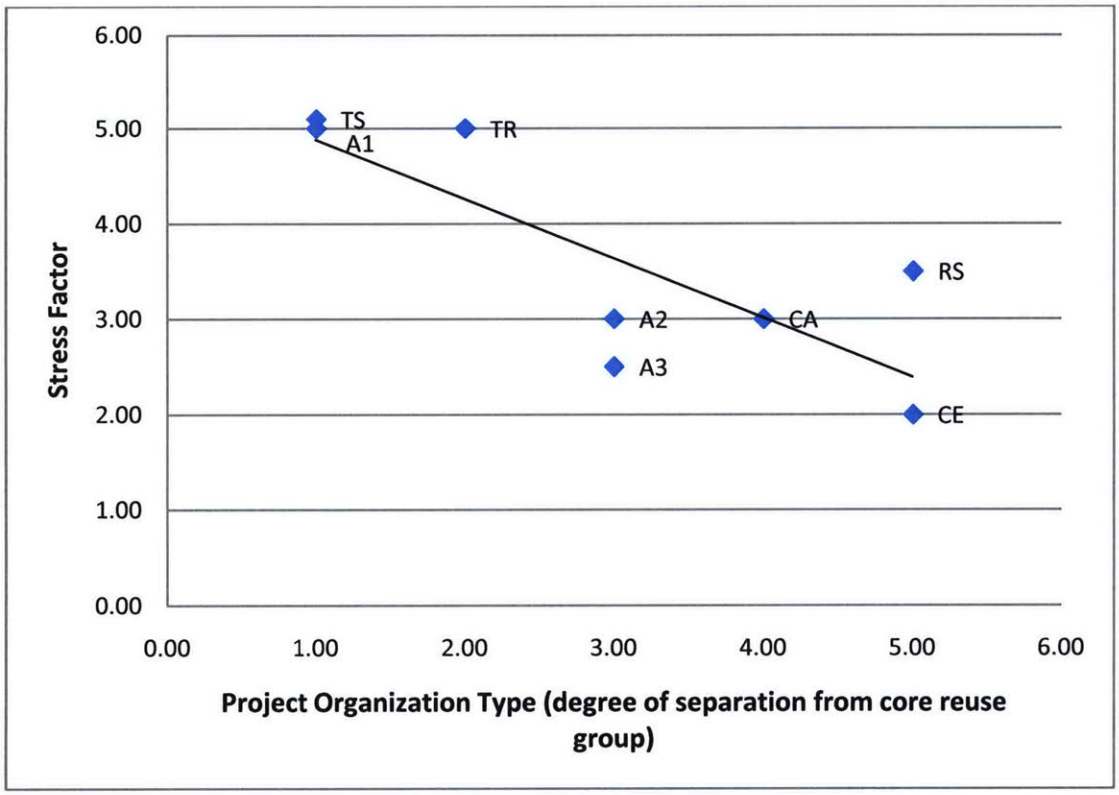


Figure 22: Stress Factor decreases with team independence

As we see from Figure 22, three of the four high stress factor cases (TS, A1, TR) were common product development. The fourth (RS), was a client project, but performed very early on in the reuse program's lifecycle. This is illustrated by Figure 18, above, and shows that RS was only the second reuse project executed in the organization. The theme here is that TS, A1, and RS felt real pain during execution because they were so early on in the program's lifecycle. It is also worth noting that at the time of RS, the hybrid option for teams did not exist in practical terms, since management pushed teams to one extreme or the other. The lack of documentation and client support made it very difficult to be a client project at that time. The overall immaturity of the product made it very difficult for a common product development team to execute smoothly soon after the time of RS, especially under the pressure they felt to get the common product launched while meeting customer requirements and schedule.

System Performance is an important concept both as an independent variable driving the effort to find a solution, and as a less tangible quantity affecting the team as they are challenged to solve the problem. In this second manner, required system performance

is a very important factor, as it affects a development team strongly. Many developers can insulate themselves from pressure by management, regardless of the frenetic pace of a poorly planned project. They find sanity in the simple "do or do not" aspect of trying to solve an engineering problem. When that problem proves significantly challenging though, even seasoned developers will feel pressure to deliver, and that is when the stress factor goes up.

Although stress factor is too vague of to be an independent variable, it reveals an important insight when used in the second order to illustrate the effect on cost performance. Its negative effect on team productivity, coupled with the multiple intangible effects on groups that interface with the software team serve to hamper the cost performance of stressful projects. Figure 23, below, shows this relationship rather strongly. Since CPI represents performance over hundreds of tasks and thousands of hours, a couple tenths in CPI on a large development project is significant.

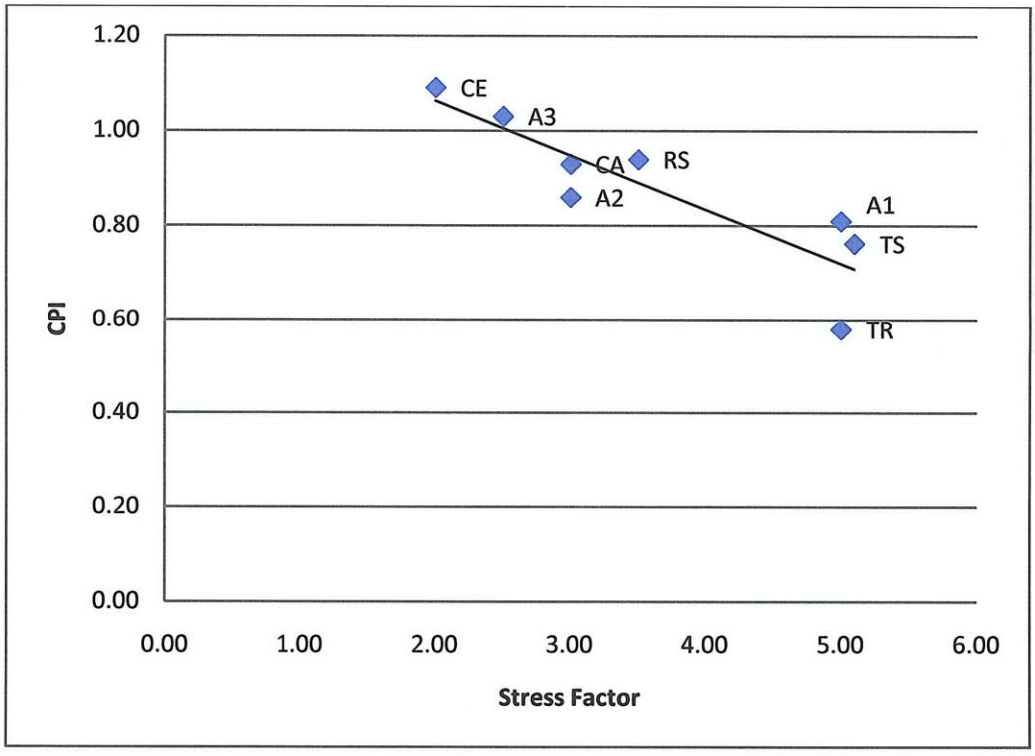


Figure 23: Cost performance also suffers with Stress Factor

Productivity is a very important metric to software managers across many industries. It indicates how well the developers are performing on an objective scale, and is usually measured in Lines of Code (LOC) per day. Though absolute scales vary for different software languages, relative values taken in longitudinal views within an organization can show how a team is performing compared to earlier efforts. In this study, we may question management's rationale for leading a team along a particular reuse plan. In this case, it is helpful to look at productivity as well as performance against the plan to normalize how well the team was doing regardless of how aggressive the plan was. This is shown in Figure 24, below, where the highly leveraged projects (A1, A2, CE), do not illustrate high productivity. The projects that pleasantly surprise us with great code writing performance are those that had low expectations stemming from mandated reuse. TR and A3 did, in fact, beat their planned reuse factor by 59% and 23%, respectively. This shows that high reuse level is possible, but demonstrated more often when management doesn't push it. This could mean that the team leadership is not particularly effective at driving the team to effective software development via reuse.

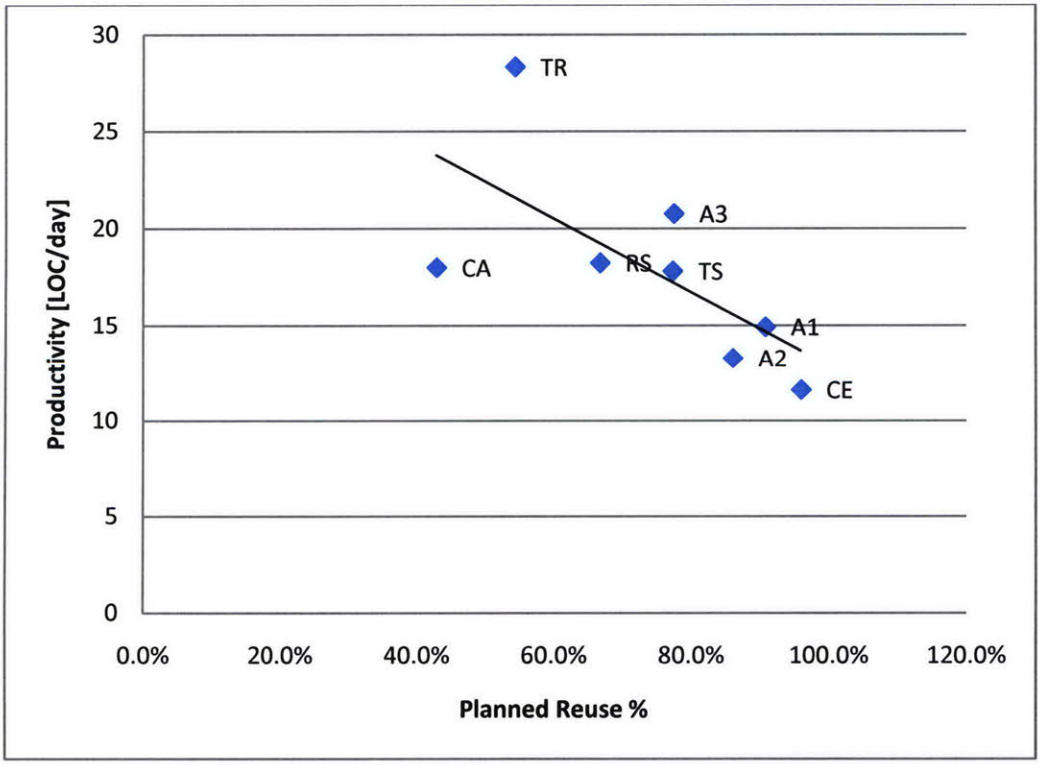


Figure 24: Less mandated reuse for higher productivity

4.4 Cost Models

Sooner or later in every cost reduction or process improvement initiative, we must answer the question of how well the investment is paying off. In this case, one answer lies in the application of a cost model, like those discussed in section 2. The Gaffney and Durek model[29] used for our subject organization shows that all Odin reuse projects considered have a cost factor, C, of less than 1. C is "the cost of software development for a given product relative to all new code (for which C = 1)". For each of the projects we have discussed, C is shown below in Table 6. Of the two models shown, C2 – the *cost of development* model, is more realistic as it accounts for the added cost of developing for reuse and amortizes the benefit over a finite number of consumer projects. We chose a 30% factor for the cost of incorporating reusable software into a new baseline, and a 20% added cost to develop code as reusable. For consumer projects (n), we identified the known candidate projects either planning to reuse the code or similar in requirements that their likelihood of reuse was high. We then added one more project to the model for a downstream consumer that we didn't know about.

Table 6: The Gaffney and Durek Model shows significant cost reductions

Project ID	C1 (G&D)	C2 (G&D)	n
T2	66.6%	78.1%	6.00
T3	45.9%	64.5%	5.00
A3	39.8%	60.4%	5.00
A4	45.8%	69.0%	4.00
RS	53.3%	73.3%	4.00
CA	70.1%	95.7%	2.00
CE	32.8%	90.4%	2.00
TR	62.0%	75.1%	5.00
A1	36.5%	54.6%	6.00

[29] Gaffney, J. E., & Durek, T. A. (1989) Software reuse – key to enhanced productivity: some quantitative models. *Inf. Software Technology, 31, 5, 258-267.* as cited in Frakes & Terry '96

We see that there are only a handful of projects (shaded) with cost factor (C2) below 70%, which shows that the reuse initiative was an acceptable investment. With higher estimates for n, reuse consumers, these numbers are much more favorable. It is notable that some of the better performing projects along other dimensions, like CE, don't show that well here. In this case, it's because CE only 'feeds' two other similar projects, as its content is somewhat esoteric.

The preceding analyses have led us to conclusions and recommendations about team design, stress factors, and cost performance. Those insights are synthesized and included in the next two sections. Although this work may serve to improve the reuse effectiveness of the organization, a measure of cost effectiveness for the Odin program has already been performed. The money saved over just the first 4 years of the reuse library's existence was surprisingly high, as shown in Table 7. The total amounted to (roughly) 56% of the business unit's annual sales. These numbers should speak for themselves, showing that each project's software effort was reduced by (on average) 80% from what it would have been if there was no reuse repository from which to draw and the system had to be created from scratch. Certainly, this is a significant success for the reuse program. Unfortunately, competitors were employing similar techniques, so the focus would shift from launching the reuse program to sustaining it.

Table 7: Total accrued cost savings by the Odin Software Reuse Program

Project ID	Odin LOC Bid for Reuse	Bid Total LOC	% Odin Reuse of Total LOC	Estimated SW Reuse Savings (rel. to Avg contract size)	Estimated Contract Reuse Savings (% division annual sales)
A	92,847	145,542	63.8%	87%	6%
B	113896	143436	79.4%	107%	8%
C	70097	97475	71.9%	66%	5%
D	122551	191094	64.1%	115%	8%
E	78650	159853	49.2%	74%	5%
F	69428	83548	83.1%	65%	5%
G	73210	83350	87.8%	69%	5%
H	31570	74973	42.1%	30%	2%

I	147590	191024	77.3%	139%	10%
J	55141	109055	50.6%	52%	4%
Total				**805%**	**56%**
Average				**80%**	**6%**

5 Recommendations

After searching the body of literature for guidance on software reuse, examining the sponsor's case studies, and analyzing project data for relationships, we see the following themes:

- A concerted and sustained reuse program is not a trivial undertaking. It requires an earnest and sustained executive commitment, investment, a solid architecture, and execution that never loses sight of the reasons for having a reuse program.
- The cost/benefit balance can be looked at a few different ways, but an organization must obviously make its investment back and aims for the ROI to be as high as possible.
- A reuse effort takes time to grow and mature. Early projects will see challenges and successful execution does not become more easily attainable until later, when the process and the repository are mature.
- Through the large number of reuse projects studied, a common theme is that the human factor is often a major cause of lack of adoption or full success of a reuse program.

5.1 Earlier Hypotheses

By simply examining project data in section 4, our hypotheses were fairly narrow when drawn directly from the data. They are listed below, along with a corresponding updated hypothesis that is broader.

Hypothesis	A Larger Implication
1. The early projects in the sponsor's reuse program fell below expectations due to the immature reuse repository and process.	It takes time to make a reuse program effective. The organization should be fully aware of the path they are on and set expectations accordingly.
2. High stress factor projects will have	Highly leveraged projects often exhibit low cost performance, possibly due to decreased

poor performance	productivity.
3. Overly high expectations and mandated reuse strategies will negatively impact the development team.	Productivity and performance suffer when a project's development team is directed to reuse too much or create software with too many reuse consumers.

5.2 Updated Hypotheses

By viewing the preliminary data-analysis-based findings through the lenses of the industry literature and survey responses, these hypotheses can be refined, with some new observations added.

Although the literature suggests that the architecture is the technical part of reuse that's vital to get right[30], the sponsor's experience suggests that the next decision – what portions of a product platform are common – is just as important. The scope of reuse on a product should match the commonality of the customer's requirements. If the use case and requirements of a system is so different than its previous applications, those parts that are different should be considered for exclusion from the common reuse product.

'Fishbowl' projects carry extra baggage. The analysis of project data shows that highly visible, highly leveraged projects cause an increased stress factor on the development team. This, in turn, can negatively impact productivity and cost performance.

The cost/benefit model[31] helps illustrate the situations that will provide the highest payoff for the reuse effort. Although these may seem obvious, it is important to remember that software reuse is only effective when:

- The incremental costs of implementing a reusable component is low
- The cost to reuse a component (integration) is low
- The payback benefit from a large number of reuse instances is high

[30] Jacobson et al, '97
[31] See the application of the Gaffney & Durek model in section 4.4

The hard choices made during the proposal and planning phases usually represent cost cutting by assuming a task can be done with a less expensive approach. These assumptions and adjustments in approach must be carried through to the execution. If the total funding available for a proposed effort is less than the estimated cost for the solution developed by the engineering team, then cost trimming measures are sometimes taken. These steps can cause changes in the technical baseline of the project to an approach that is easier to implement (and thus saves money), but suffers in some other measure (like performance). The change in the approach may save money, possibly by using an older, less capable version of the product platform while still meeting the requirements. Any changes in approach to other portions of the reuse library must be communicated to the development team as new direction so that the project can execute within its new expected budget.

The business model for software reuse in the defense electronics industry requires some extra thought. The fundamental problem is that each customer and contract expects to only pay for the work done that directly contributes to their deliverables, but also take delivery of as capable a system as possible at the end. Additional capability for the same amount of funding is enabled by the ability to reuse portions of the system from prior efforts. Customers see this as a discriminator when selecting the contractor. So, no one wants to pay to make software or products reusable, yet everyone wants to take advantage once they are developed. The complication in the defense industry is that customers pay for most of the development, and they can direct what their money is spent on. It is interesting to note that the customer funding issue may only be applicable in the long term, steady-state business model. Initial development may be funded with internal R&D dollars, but the reuse model must sustain itself by paying for enhancements, maintenance, and some upgrades. That sustainability is the real challenge to the business model, and its cost must be borne in some way.

5.3 Summary of Recommendations

The sponsor of this research executed a foray into software reuse for roughly 10 years. As we have previously discussed, that effort was well thought out, architected, and designed. It progressed from simple software reuse in a component-framework

architecture to full system product platforms. The challenges faced by the sponsor organization raised questions about how reuse might be better executed in the future. Of the two research questions, the second remains unanswered: "What a prescriptive framework that will help steer future projects?" Revisiting our graphical view of the situation, in Figure 25, we note that it's really the roadmap that we're searching for.

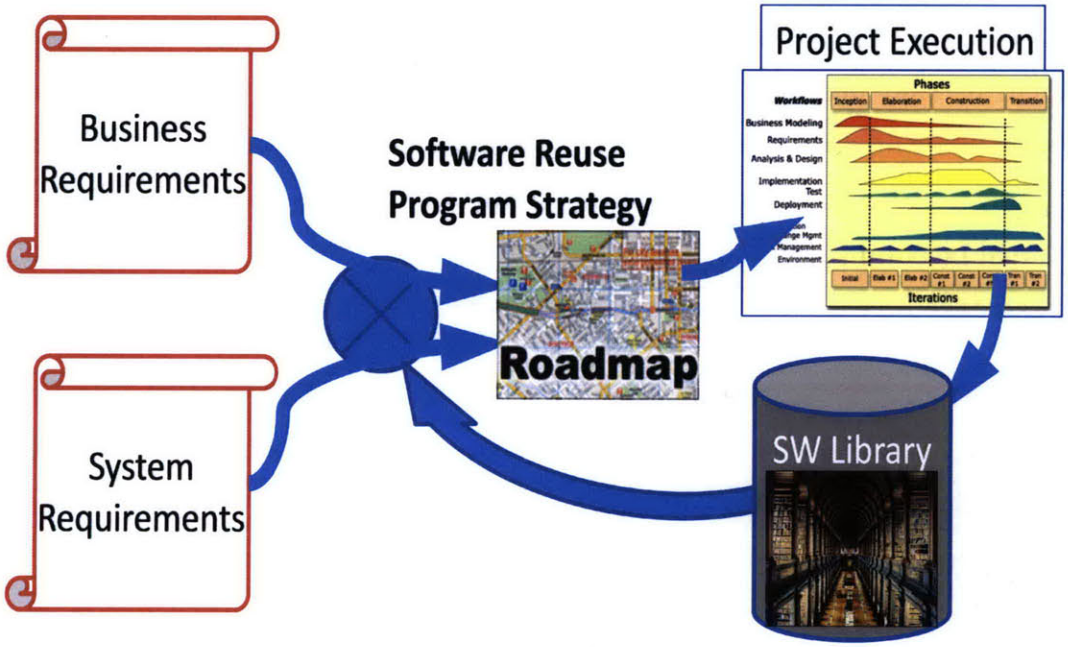


Figure 25: Multiple sources of requirements shape the execution roadmap

A way to easily identify a project's roadmap is sought. This framework for software reuse can be prescribed to the client organization and in other relevant subject areas. First, we look at the inputs and constraints, and ask what guidance can be given to the decision maker. If we take a straightforward approach to developing the project's roadmap, we note that identifying the primary driving influence is vitally important. This is similar to developing a value proposition for the project, which identifies the value being created in by its efforts and the ranked importance of those valuable outcomes.

Table 8: Driving Constraint-based framework

Driving constraint	Ideal Org type	Aim to reuse...	Recommendation	Expectation to Set
Cost	Client or Hybrid Team	As much as possible	Aim for high reuse – be a consumer. Keep the distractions out	Delivered system will be just like project Q (an earlier one), with only minor changes
Schedule	Hybrid Team	An amount appropriate to low risk	Ensure technical baseline is bounded to pieces of the reuse library that can be assembled with low risk of reiteration; Keep the distractions out	We only have time to do it once – ensure a low-risk design
Product Platform; design/implement for reuse	Core Reuse Team	As much as possible	Don't accept too many new requirements Ensure 20% extra cost of 'making it reusable' is included.	System is part of a product line; many influences will appear
Scope or System Performance	Client Project	Only what is already proven to meet the requirements	Only reuse components that have proven their ability to meet the performance requirements Consider (at least some) new development.	Departures from the common product may be necessary

Even with a framework like this, project leadership must fight the tendency to say that they have all of the constraints listed. At most, two should be identified as being "driving constraints". In addition to the model for project organization type and expectation setting, many other factors will influence the sponsor's ability to practice large scale reuse. Product platforms or large scale software reuse in the defense electronics sector is challenging and only a good fit if you have the right conditions – the following are necessary conditions we've discussed earlier in this work:

a. The project must have a customer that's amenable to tailoring some of his requirements to match the direction of the common product.

b. The reuse team must create an architecture that supports reuse.

c. For a project to be cost effective, it needs a software reuse library that's mature.

d. The organization must have a reuse process that enables good citizenship consuming of reusable components or subsystems and returning them to the repository. "Clone and own" reuse and branching is easy, merging back is the challenge that must be mastered.

e. There must be enough projects to amortize the cost of development and maintenance. A small number of projects will not justify the use of product platforms.

As evidence of programs demonstrating these conditions, we can contrast the reuse examples in Table 9. This broader experience base from other reuse efforts within the enterprise shows some key points that come with different kinds of reuse programs.

Table 9: A Comparison of reuse programs

Example of reuse	Key Benefits	Size of reuse community	Limitations
Product platform (generic)	Substantial R&D benefits from well designed platforms[32]	A software development organization	Nothing beyond the scope of the common product
Odin	Strong component-framework architecture; mature reuse repository	A business area and related government research labs	Shouldn't grow beyond span of understanding the architecture
User-forge	Common-man accessibility, more flexible	Grass-roots	Doesn't provide reusable subsystems, just modules of code
EW system "example A"	High performance	Single software development team	Not designed for reuse; extensibility limited
Intelligence Community "example B"	Business Model proven more sustainable thus far	Larger; developers already using a common language in the community	Shouldn't grow beyond span of understanding the architecture

[32] Meyer and Seliger, 1998

In addition to the framework above, the difficult question remains of how to generate sustainable reuse funding. This analyst's recommendation of a primary source is to capture this cost and pass it on to the projects through modest license fees. What might seem like a trivial amount for one project could really add up once the reuse community grows with numerous customers and contracts. The sponsor's organization never charged for licenses, nor attained critical mass of income to offset this cost. Another source of funding is to identify the software library for one of its uses: it is a technology push in some markets. To support a push model, research and development customers must be found, sometimes for the sake of advancing the technology. A deeper and long-term relationship with Customer Research and Development (CRaD) sponsors is in order.

If these recommendations prove fruitful and some of the specific conditions are met, the sponsor will achieve what few in their industry have: a large-scale, long-lasting, effective software reuse program. Not only will it save the company and customers money, but ultimately the taxpayers as well.

6 Conclusion

This work has tried to examine software reuse in challenging business and technical environments. Although the subject of software reuse is not unique to any one industry, defense electronics includes a unique combination of business and management challenges with stringent system functional and performance requirements. In this arena, we see an incredibly high need for dependable systems to help bring home safely the men and women that are put in harm's way in defense of our country.

Other areas with similar constraints are: federal contracting for FAA and NASA. The difference between these and the commercial world of high tech computing, or the automotive industry is that the need in defense is to solve a tough problem that no one has solved before. Meeting this need is motivated by the inherent responsibility to equip our warfighters with systems that keep them as safe as possible. To do this, the tough problems must be solved quickly and fielded expeditiously. The motivations in the commercial sector are often more focused on time to market and ultimately profit. Although profit can be an extremely powerful motivator, it may not percolate throughout an organization in the same way as protecting our warfighters does.

The sponsor of this research is a large defense contracting organization with many divisions, sectors, and business units. The subject business unit, in particular has offered a colorful experience base from which to analyze software reuse as it applies to developing systems to solve hard problems.

The subject of effective software reuse in the defense industry is intriguing and interesting from both an academic and practitioner's perspective for the following reasons:

Reason	Why it's unique	What it means
Products are complex electronics systems for unique purposes	System functional and Performance requirements are different than commercial	Software developers are challenged to solve hard problems.
Government push for	Solving hard problems with	Difficult requirements can

'better, faster, cheaper"	ever-decreasing resources (money)	be met with enough time and money. But both are also constrained
Projects primarily customer funded	Business models have been typically designed around outside funding sources	Internal funding is scarce; applying product platforms with commercial business models is not a perfect fit and must be rethought

To explore each of these unique attributes is to gain understanding of the influential forces that shape execution and outcomes in software reuse programs and the projects of which it is comprised. The inherent *complexity of systems* designed in the defense electronics arena flows from a need to keep systems:

- Small and light (increased endurance of an aircraft)
- Very or multi functional – having two similar systems when one can be 'extended' to do both is a waste of space, money, and time. Most, if not all systems must meet hundreds or thousands of requirements. They are simply more capable than an iPod.
- High performance – missions performed in our nation's defense must be capable of thwarting an adversary and bringing our warfighters home safely. Systems must therefore perform their functions to the highest performance level possible.
- Seemingly un-complex to the user and without need for extreme training, logistics, and support.

For these reasons, systems are often made of custom or semi custom hardware with increasing levels of software doing the hard math, heavy lifting, or higher intelligence. As we discussed in section 3.2, software systems have been used to maintain hardware footprints and interfaces while allowing for upgrade paths.

Cost Pressure – Defense contracting is simultaneously a customer funded development and continual cost pressure environment. This stems from the public scrutiny under

which the budget is viewed and movements within the government such as "Better Faster, Cheaper" of the late 1990's.

Funding sources – While the positive light of customer funded development shines on software projects, we must examine the business model. With only 1-3% of sales devoted to R&D, and then only for things expected to develop real new breakthroughs, there is no 'bucket of money' for operations and maintenance.

6.1 Findings

Reuse strategies need strong executive support and a robust funding model. Almost every article written about software reuse programs cites the need for executive commitment. The need stems from the support required to secure significant investment dollars to create the core software architecture and prototype repository. The sponsor organization began with roughly $20M for these purposes and wrote the first 100,000 lines of code. That body of code had no further funding earmarked for its care and feeding in the first few years. One key uniqueness of the defense industry and others is the challenge to fund maintenance, bug fix, and tool evolutions. It is a fact that:

- Software that's used in slightly different ways will exhibit different behavior and sometimes reveal flaws and uncover bugs.
- Moore's law keeps technology (especially hardware) moving forward at a rapid pace. Platforms (hardware), tools, and networks evolve and pace must be kept.
- SW vendors are in business to make money and produce new software (versions, that improve upon the previous) to do just that.
- Improvements are made in the way we interact with our world through software, and those should be adopted by defense electronics if it makes the warfighter's job easier and brings home more users safely.

We have shown that meeting these challenges with the classical defense funding model is difficult. With the adoption of the 'better, faster, cheaper' movement of the late '90s, the industry saw an end to the fabled large, gravy train projects for well funded and

tolerant customers. And so, despite being paid for development work, a reuse program must incorporate an operations and maintenance funding stream into the organization's business model. In section 3.3.2, we saw that the license fee structure has proven successful for another reuse program within the company. Still other groups inside and outside the enterprise use customer funded R&D as a source of the funding stream if they can convince their customer that they are moving the mutually necessary technology forward.

Reuse team design should be driven by the strategic objectives of each project. As projects vary throughout the product platform portfolio, so do their organizational needs. In section 3.3.1 we discussed three team types (core reuse, client, and hybrid). In section 5.3, we proposed a framework for making the choice of team type as well as the design drivers for the project. These choices are vital to a manager starting a new project. In sections 4.2 and 4.3, we saw that projects suffer if saddled with cost or schedule pressure along with a core reuse objective. Major common product work must account for the 10-20% increased cost of commonality, and those project teams will feel the scrutiny of living in a fishbowl for execution. We conclude that projects without a strong need for strategic reusability be allowed to execute as hybrid types so long as the organization can account for management of their software baseline after completion. If that baseline is to become more common, either enhancement or new contract funding must be obtained to bring it in to the product platform.

The scope of the Product Platform should be limited to what's consistent from project to project. A product platform approach must think very hard about the scope of the reusable product. Making a well considered (and correct) decision is the first step to success. As Jacobson's cookbook for software reuse says, "reuse depends on architecture". Though our DSMs show that the Odin architecture is very modular and suited to reuse, we know from 2 projects in particular (TS and HY) that some software 'modules' were not common enough to be reused. These were upper layers of the architecture, close to the user (tasking and UI's), and resulted in rewrites of existing code that would've been easier if the reuse of those layers had not been attempted.

Those layers that can change with the use case should be designed to be very thin so their development as one-off portions can be as short and inexpensive as possible.

This is not to say that the product platform is an incorrect fit. We see commonality between the classic platform advice from the DoD acquisition community and the SEI, "... especially given shrinking defense budgets, it makes no sense to build essentially the same systems over and over."[33] ... and from within the core team of the sponsor organization: "everybody thought they were building something for the last time every 5 years."[34] The repeated implementations came to a stop with the product platform, but the higher layers must be carefully architected in or (probably) out of the common product.

Project Odin enabled the subject organization to stay competitive and saved an average of 80% on each project. The project was launched at a critical juncture for the business and required significant investment. The cost/benefit models discussed in sections 2 and 4.4 show that it saved the large amounts of money on future projects that were needed to keep the business competitive. We have discussed that creating and maintaining a sustained line of product platforms or large scale software reuse in defense is challenging and only a good fit if you have the right conditions. Detailed in section 5.3, these conditions are met by the sponsor organization and will serve them well in the future of their reuse program.

Closing comments: We have considered some seminal works on software reuse by Kemerer, Poulin, and others. These broad and longitudinal studies provide enough data points to make strong general statements about software reuse in the industry as a whole. They do not, however, include cases from the defense electronics industry and so this work has attempted to project their insights into a new space. We acknowledge that the best way to study this subject would be a comparative study of head-head organizations over many years with similar projects. Though the value derived by such a study would be great, the possibility of executing it, with disparate corporations in a

---

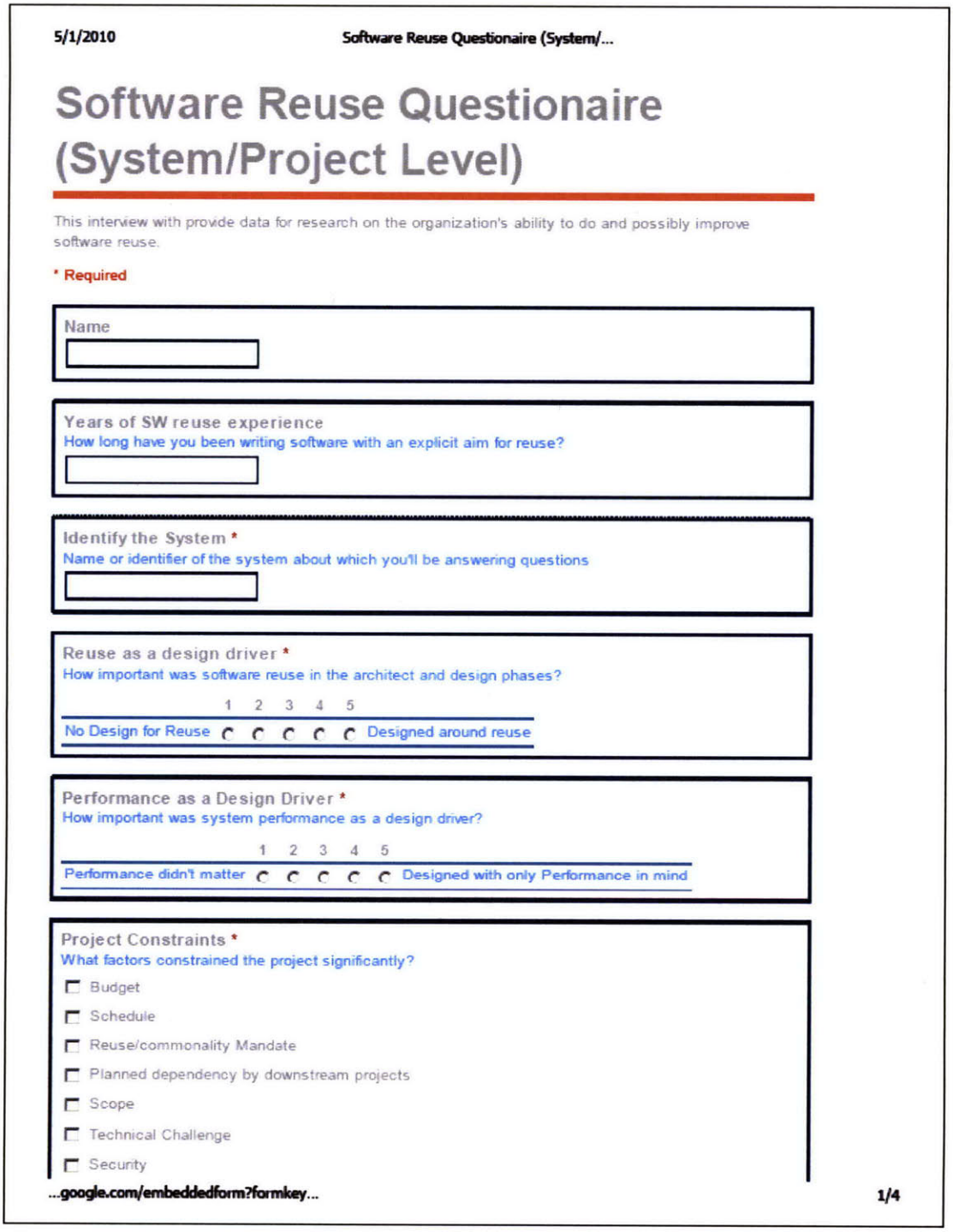
[33] Linnehan, J. (2005) Office of the Assistant Secretary of the Army. As quoted in DoD Software Product Line experiences: A Digest of Participant Presentations. *The 8th Software Product Line Workshop,*
[34] Interview quote, subject #3

competitive landscape is nil. Without such an effort, this work may serve to provide similar insight, at least to the sponsoring organization, and enable effective software reuse in the future.

At the highest level of observation over many software intensive projects, projects can be modeled as having inputs, managerial decisions, execution factors, and resultant outcomes. They take inputs in the form of programmatic constraints, system requirements, and an existing software library. Each project's combination of influences and constraints makes it more suited to one of a few different models of software reuse, and the manager's challenge is to choose the best fit to enable success of his project and the business overall. We have developed a framework to guide the manager in designing projects for overall individual success and contribution to the product portfolio. Hopefully, this work will serve to 'tighten up' the organization's reuse program and provide insight into how its valuable portions of solid architecture, process, and team design can be leveraged into long term sustainability.

Appendix

A1: Interview Instrument



5/1/2010 Software Reuse Questionaire (System/...

Software Reuse Questionaire (System/Project Level)

This interview with provide data for research on the organization's ability to do and possibly improve software reuse.

* Required

Name

Years of SW reuse experience
How long have you been writing software with an explicit aim for reuse?

Identify the System *
Name or identifier of the system about which you'll be answering questions

Reuse as a design driver *
How important was software reuse in the architect and design phases?

	1	2	3	4	5	
No Design for Reuse	○	○	○	○	○	Designed around reuse

Performance as a Design Driver *
How important was system performance as a design driver?

	1	2	3	4	5	
Performance didn't matter	○	○	○	○	○	Designed with only Performance in mind

Project Constraints *
What factors constrained the project significantly?

- ☐ Budget
- ☐ Schedule
- ☐ Reuse/commonality Mandate
- ☐ Planned dependency by downstream projects
- ☐ Scope
- ☐ Technical Challenge
- ☐ Security

...google.com/embeddedform?formkey... 1/4

Your tendency towards reuse *
Given a design/implementation task, you seek to reuse software how often?

	1	2	3	4	5	
never	○	○	○	○	○	always

Degree of reuse *
You seek to reuse software to what degree?

	1	2	3	4	5	
code snippets	○	○	○	○	○	subsystems of components

Suitability of SW library for reuse on this project *
Did the proposed architecture have a solid foundation in the SW library as it existed at inception?

	1	2	3	4	5	
No, we knew we had to start fresh	○	○	○	○	○	yes, many components already existed

Type of SW reuse *
which style of reuse was employed by management

- ○ Nothing formal, just get it done
- ○ Reuse by employing reuse-experienced individuals but on a team separated from the common product group
- ○ Work done primarily by a Product Centered Organization
- ○ Other: [ ]

Key Components *
Which "Components" and other major software modules (services, HMI) were key to the systems functions and performance?

[ ]

Achieved software reuse *
How successful was the software reuse initiative on this project?

	1	2	3	4	5	
unsuccessful, we scrapped it all and started from scratch	○	○	○	○	○	Very successful, according to plan or better

Achieved System Performance *
To what degree would you say the system met its performance requirements?

	1	2	3	4	5	
fell on its face	○	○	○	○	○	totally nailed them

System Performance challenge (Technical) *
Rate the overall challenge of this system compared to other systems in your career

	1	2	3	4	5	
a yawner	○	○	○	○	○	The hardest technical challenge you've ever seen solved

Overall Success
Was the project a success? by what measure? was it a failure on any dimension??

Data about project
Enter anything specific you know about the project: Task Code, target/achieved productivity data, kLOC, reuse %, SW hrs., rework hrs, etc...

Further Contact
May I contact you again if needed for follow up?

- ○ Yes
- ○ No

Recommended other sources
Who else should I talk to?

A2: Interview notes and quotes

What follows is a compilation of interview notes. [Text in brackets indicates paraphrasing for clarity]. Highlighting indicates (possible) inclusion of this quote in the main text.

Interview Subject #1

Position: Systems Engineer and architect of reuse project

We had two choices: either do what the project wants and meet their requirements, or [**hold firm to a product vision** that may not meet their requirements]. We can be responsible for the portfolio to the [Vice President/General Manager].

There are two ways to look at it: 1. Success is the sum of the success on projects, or 2. We should expect success to be the non-linear aggregation of the projects.

[initial ideas came from] the Object Management Group. [we considered the] CORBA component model. The original idea was not to invent a framework but to buy one.

We were realizing that now (c. 1999) we can do in software what we couldn't before. The CORBA framework model didn't gel in time.

[The reuse program] started with the idea to be a toolkit provider. But they (client programs) weren't using the tools the best, so we decided to provide more.

Have to have enough critical mass... we **never figured out how to run the business to [support the reuse program]**. Maybe you have to feed the beast. Do some things to simply advance the technology... to product platforms need. Need to sell projects both ways – solve customer's problem *and* support the technology push.

product platform approach requires an executive commitment

[Another idea for the financial model is to take some funding from each project and use it to fund the maintenance effort. You can do it if you declare that in your rates to the fed govt. we never did this in the reuse project. The reason was probably to maintain the same rates across the company. The division with the reuse program was not all that large.]

Interview Subject #2
Position: Business Area Technical Director

As the "father" of the reuse program, this interview warranted a departure from the standard question set.

Q: Why the component/framework model?, why was that the vision for the reuse model?
A: large scale SW reuse was the whole idea; we needed a modular architecture in SW similar to modules in HW
It needed to support **HW independence, scalability, and a very good communication fabric** (3-4 tiered model)
[the principal contributors] started **reading like crazy**... they considered RUP
We quickly realized that we needed a comp-fw model...
[Looking back], just **OO was largely a failure**
Settled on a framework on top of CORBA (almost used corba)
We wanted a well-accepted open standard, this would meet customer pressure for open-ness

wanted to change things without rebooting

Q: how did the software reuse and common product organization evolve?
A: [Eventually, it] was a support organization; [the SW itself]**it aint really all that portable; so let's build more hw/sw integrated**
We didn't have enough distance and rigor to avoid designing special stuff into the core."
[we would] build subsets of what we should've built, not the whole thing, and not fully tested b/c all they wanted was a piece.
We started to design waterfront of what all customers would want.
We argued that we needed to maintain certain amount of IRaD [funding], [otherwise we would have] no maintenance funding every year. [maybe we] needed to charge maint on contract

[Another group in the company] has a very different SW reuse model: project X starts w baseline, adds features, but they can't spin it back in, so the next project pays to integrate it back in.
Another group is model based [Like the Rational Unified Process OO model]. It's faster [to change the model] from scratch than if they tried to reuse the code. They wanted to keep model reusable

Q: what kills the product platform?
A: [One aspect might be] **Culture/background... [Theory that] young people might be better at not hacking through the code; We need a business model that supports general reusable core and reuse it on a bunch of programs.** [We need to learn] **and how to maintain it and maintain the programs.**

[Maybe a] business model that's not product based, but program based. Use CRC: classes, responsibilities and collaborations?

[Near the end of the design phase, we] **had a bulletproof architecture. Couldn't stump ourselves**
but then we had to build it. **Needed a really strong vision to ride herd**
Maybe the component/framework model wasn't robust enough to scale?
We didn't tier out a structure that made it easy to scale.
Maybe we weren't big enough to take independent hw...
(A competitor), Argon glomed on to mercury, who had a value added.
[We got all confounded about] the IP.
But we could never really say that this sw is open, b/c we kept tying it down.

[In order to have a sustaining business], you gotta have critical mass
But, we can't sell tech to pilots; maybe we weren't close enough to customers.
Also, we didn't show rapid prototyping

Interview Subject #3
Position: DSP Software Engineer

We were challenged by reuse of stuff not necessarily meant for reuse
I usually reuse mostly at the cut and paste level
OO was too hard to understand, hurts maintenance.

Every five years someone says "I'm gonna write this for the last time"
Overall, the OO benefits were not realized
The viability was challenged by lack of talking to users.
Our goals were too high given what we might have seen what the company commitment was going to be and when it was going to stop.

Interview Subject #4
Position: Project Manager

When we started, everyone was new at it. We didn't know well how to design for reuse
For instance, how big should a component be?

We didn't know how to document reuse? Component Data Sheets fell behind
It was hard to decide if guis are reusable. What shouldn't have been reusable? How to make good subsystems?
We feared we tried to make too much reusable at the higher level
We should've stopped at component, tasking, gui pieces
It's easier to sell 20 pens for $1M than a few with $5M customization
We would need to pay I&T anyway, and should've let it happen
Customers always come in needing new stuff
look at ESU for that model.
Yes, we didn't 'feed the beast'
So, we had no other place to get $ for maintenance

Interview Subject #5
Position: Software Manager

"We do ok at reusing sw, it's the giving back that we do a poor job at"
creating a repository is hard - easy to find, identify, and reuse
[the grassroots movement within the company called xfg is an] example of smaller scale reuse.
It is a big step forward
The ideal is something like linux – [it's open and reusable, but when you install it, you] expect to work
The most often [pattern used] is when we take the algorithm out, build around it
We have a hard time b/c we don't feed it back in and make usable by others

Interview Subject #6
Position: Software Architect

"The toughest thing about reuse is maintaining synchronization to a baseline - so that existing programs can take advantage of changes by future programs"
The best reuse level is components and libraries, it's less about subsystems.
You need to reuse on a module scale to consider it reuse.

Interview Subject #7
Position: Software Engineer

Communication or lack thereof on a technical level handicaps us
Functional breakdown contributed to communication issues (why re-writing tasking in java?)
Biz model wasn't consistent and realistic. Didn't balance the needs of the program and needs of a product. We needed more standardization. Treat people as one group. A Tech steward is the team lead. Too often people are pulled as warm bodies.
We tried to foist cost on programs (merging); need some coordinated internal funding; lay out common work at planning stage, get it paid for.

Interview Subject #8
Position: Business Area Software Functional Manager

At least Odin was designed for reuse; as opposed to forcing reuse of something not [designed that way] like EW example A
[More often we see] clone and own as opposed to reuse program
If you're gonna do reuse, everyone's gotta be on board with it (SW, SE, HW)
Do we allocate reqts to reuse items?
Which gives a competitive advantage for bidding/executing programs?
Q; Shouldn't every large company have a gforge type repository?

Interview Subject #9
Position: Project Manager

What did they want to reuse? Algorithms, HW isolation
Also discussed: modularity (boundaries, robust interfaces) in a(n iron?) triangle with NRE and performance.
Now we're reusing components, tasking, and subsystems.

Interview Subject #10
Position: Software Manager

We're not realistic about what we expect to reuse and then actually reusing it
We don't **communicate what the expectation is based on the sw bid** spreadsheed
CA was a system reuse thing – exercised new paths in the code, of course you're gonna find bugs.
CE: reused exactly what we planned. Took some particular people to get it right (that had learned from CA?) and stick to the plan
Yes, the reuse program is a good thing. Stick to the plan. Need better tools for sharing and a better understanding about what we have (ref Jacobson here)

Interview Subject #11
Position: Software/DSP Architect and Developer

We need a pull model. Each group pulls some good stuff out. We have 'keepers' that maintain the latest and greatest of the 30 main things. (Interviewer response) there's no guarantee they'lll all work together.

We shouldn't try for the full nut: tasking, components, GUI; but rather something more modest – components of parts thereof.

Interview Subject #12
Position: Enterprise Director of Software Engineering

Developed a mode for cost avoidance... reuse still requires 35% more to integrate and test it. EW example system A, there was no substantial investment in trying to execute that reuse strategy. Needs to be unraveled and rewickered as a framework. It's not a SW architecture, the software is there to make the hardware work.
IC example B is a big success. In 16 countries with 1000's of users. It was productized and looks forward into the market. Licenses are sold in industry, even within the company. Therein lies the funding stream. Charge less, build momentum.

A3: Acronym & Definition List

ACTD – Advanced Concept Technology Demonstrator - a type of DoD program, early in the lifecycle.

CPI/SPI – Cost Performance Index, Schedule Performance Index

Component – a software module that plugs into a component-framework architecture.

DoD – US Department of Defense

Framework – a piece of software that manages communications and isolation of components in a component-framework architecture.

IRaD – Internal Research and Development

NPV – Net Present Value

LOC – Lines of Code

ROI – Return on Investment

RUP – The Rational Unified Process – a process for iterative object-oriented software development, developed by Rational Corporation.

SDD – System Development and Demonstration – a type of DoD program, mid-way through the lifecycle.

Thread – a set of components that runs in sequence (a subsystem) to take an input and create an output. A logical flow of processing that can 'stand alone'.